2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPSGDL: An Object Oriented Graphics Description Language for Virtual World Application Support

by

Kalin Paxton Wilson

September 1992

Thesis Co-Advisors: Michael J. Zyda
David R. Pratt

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS/ZK	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
NPSGDL: An Object Oriented Graphics Description Language for Virtual World Application Support (U)

12. PERSONAL AUTHOR(S)
Wilson, Kalin Paxton

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 08/91 To 09/92	September 1992	150

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Graphics, Data Description Language, Object Oriented Programming, Virtual World Modeling, Graphics Class Library

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Many virtual world applications today represent the cutting edge in real-time 3D interactive graphics. Virtual world applications must model many complex, often changing, graphical objects. These graphical objects must be modeled both visually and behaviorly. The performance of most applications of this nature is determined by the graphics processing capabilities of the hardware used. An efficient, application independent method for describing and managing graphical objects is essential for rapid prototyping and development of robust virtual world applications. This thesis presents an efficient, flexible and extensible graphics description system, NPSGDL, used in virtual world development at the Naval Postgraduate School.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael J. Zyda	22b. TELEPHONE (Include Area Code) (408) 646-2305 | 22c. OFFICE SYMBOL CS/ZK

NPSGDL: An Object Oriented Graphics Description Language for Virtual World Application Support

by
Kalin Paxton Wilson
Lieutenant, United States Navy
B.S., University of Utah , 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Author: [illegible]
Kalin Paxton Wilson

Approved By: [illegible]
Michael J. Zyda, Thesis Advisor

[illegible]
David R Pratt, Co-Advisor

[illegible]
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

Many virtual world applications today represent the cutting edge in real-time 3D interactive graphics. Virtual world applications must model many complex, often changing, graphical objects. These graphical objects must be modeled both visually and behaviorly. The performance of most applications of this nature is determined by the graphics processing capabilities of the hardware used. An efficient, application independent method for describing and managing graphical objects is essential for rapid prototyping and development of robust virtual world applications. This thesis presents an efficient, flexible and extensible graphics description system, NPSGDL, used in virtual world development at the Naval Postgraduate School.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

The design and implementation of the Naval Postgraduate School Graphics Description Language has been a long and rewarding process. Development began almost a year ago and benefited from many valuable suggestions and a lot of support. I would like to thank those people who directly contributed to the development of NPSGDL.

First and foremost, I would like to thank my wife, Kim, for her patience and support over the last two and a half years, and Parker for the welcome diversions. This work required a lot of long days, short weekends and time away from home.

Lieutenant Alan Walters acted as a sounding board and tester for many of the concepts incorporated into NPSGDL. As one of the first users of the system he helped identifiy errors in logic and semantics.

Charles Lombardo was also an early user of both NPSGDL and the underlying class library, NPSCL. His work in ironing out errors and identifying usage issues was invaluable.

I would also like to thank my advisors, Dr. Michael Zyda and David Pratt, for their guidance and support for this project. Dave Pratt provided valuable insights into the performance requirements that needed to be addressed and gave advice on how to meet them. He also provided suggestions for features that would be useful to the applications developed in the Graphics and Video Laboratory. Dr. Zyda provided a great deal of enthusiasm and gave me complete control over the project. His previous work with the NPSOFF system provided much of the groundwork for NPSGDL.

I. INTRODUCTION

The graphical objects used in today's virtual world applications are usually complex. The typical object may consist of several hundred polygons, several lighting materials and multiple textures. Also objects may be associated together into more complex objects which support articulated and/or animated features. The need for an application independent way to describe, manipulate and manage any variety of graphical objects was recognized during research on early systems in the Graphics and Video Laboratory of the Department of Computer Science at the Naval Postgraduate School [Ref. 4][Ref. 5]. NPSGDL is the latest system developed to address this need.

NPSGDL provides two levels of application support to the developer. At the highest level, NPSGDL is a language system that can be used to describe and manage complete graphical objects as individual entities. At the lower level, NPSGDL is a collection of classes representing graphics primitives. These classes can be used directly by the developer independent of the language system. Since NPSGDL is designed and implemented in the object oriented paradigm, each of the classes that define the language system can be extended using inheritance to provide more specific application support.

NPSGDL is implemented in C++ for use on Silicon Graphics Inc. IRIS workstations. This thesis describes the design, implementation and use of the NPSGDL system.

A. PREVIOUS WORK

Graphics developers and users naturally think of graphics entities as *objects*. Thus, the object oriented paradigm is natural for designing and implementing graphics applications.

Wisskirchen has examined applying the object-oriented paradigm to existing graphics standards with success [Ref. 6][Ref. 7]. His work in providing the functionality of the GKS and PHIGS graphics systems in an object oriented framework has been used by many researchers.

Egbert, et al has developed a system that supports graphics applications at several levels [Ref. 8]. Their system abstracts the graphics primitives and rendering process to a level that is more portable and accessible to the applications developer. However, their system is primarily an object hierarchy for programmer use. It does not provide for object storage or any application independent interface.

A recent paper describes an animation and modeling system that uses object oriented techniques and delegation to provide a flexible time-aware system [Ref. 9]. Under this system, objects are represented by a list of messages. Messages have times associated with them and can be edited. This allows time varying behavior and flexible structures. The system described in [Ref. 9] tries to provide an overall framework for representing simulation and modeling involving animation.

The Silicon Graphics Inc. Inventor toolkit [Ref. 10] provides a rich assortment of graphical objects that can be combined an many ways. Inventor is an object oriented hierarchy of objects called nodes. Each node represents a specific graphical or relational property. Many nodes support high level actions like dynamic representation (different display modes) and interactive picking. Nodes are combined in tree-like structures to compose complete objects or scenes. Scenes comprise a database that is cached and optimized for display. The scene database can then be traversed to different degrees to provide flexible control and behavior. To its advantage, Inventor is designed to be system independent and provides window system and event management services. It also provides many high level objects for modeling and interactive control. However, it is primarily application based. It is a graphics library to be used by developers using the object oriented paradigm. While it does provide for storage and retrieval of objects from secondary storage, its file format is not designed to be used off-line. Rather it is designed to be used to share objects between applications. The point being that an application is needed to generate and modify an object.

Previous systems developed at the Naval Postgraduate School have concentrated on graphical object storage and retrieval [Ref. 5]. These began as binary and ASCII file formats and evolved into more complex systems. The primary goal of these systems was a standard format for object representation that could be edited off-line and shared between applications.

B. NPSGDL - GENERAL OVERVIEW

NPSGDL, hereafter simply called GDL, incorporates many of the concepts and capabilities of the above systems. In particular, GDL provides graphics primitives and higher level application support (in the form of simple animation and viewpoint control). In contrast to some of the systems mentioned above, GDL is more tightly coupled to the rendering software and hardware used in the Graphics and Video Laboratory. This helps facilitate the efficiency needed in our virtual world applications. In addition, GDL incorporates a language system that provides many advantages:

1. Off-line object definition and modification.
2. Object storage and retrieval.
3. Object sharing between applications.
4. Application independence of objects.

GDL gives application developers a high level of abstraction that supports rapid prototyping and development of complex graphics systems. Application developers can concentrate on how objects behave and are used rather than the rendering of the object. The object oriented nature of the system provides for easy extension and modification. The language system facilitates the creation and maintenance of collections of graphical objects that can be used and shared by many researchers. The simple syntax and structure of the language allows non-developers to describe, modify and maintain objects for application use without needing to be experienced developers. It also serves to standardize the representation of graphical objects across applications.

C. THESIS ORGANIZATION

This thesis is organized into seven chapters and three appendices. The first chapter is the introduction. It provides the background and previous work that the thesis is based on.

Chapter II discusses the design of the NPS Graphics Description Language, hereafter GDL. This chapter presents the design goals and motivations, the decisions made and the methodology used.

Chapter III discusses two of the high level support mechanisms designed into GDL-- animation and multi-resolution display. The simple animation capabilities provided by GDL are described and shown by example. The mechanisms and capabilities of the animation system is presented. The multi-resolution display ability allows developers to design objects that can have varying appearances or behaviors based on drawing primitive resolution. The resolution model provides two method for using the object resolutions to display an object.

The fourth chapter presents the Graphics Description Language description. The general structure is discussed and illustrated. Each of the components of the language is presented in detail with examples.

Chapter V cover the implementation issues associated with the GDL system. It details support subsystems used and the motivations for them. This chapter also discusses considerations and limitations related to C++ and the design requirements of GDL.

Chapter VI discusses GDL performance issues. Requirements and specific techniques used to provide increased speed and efficiency are presented. Comparisons between GDL and previous systems are shown.

The final Chapter, VII, presents the limitations of GDL. Directions and suggestions for future work are also discussed. Finally, Chapter VII summarizes and concludes the thesis.

Three appendices are also attached. Appendix A is a user and developers guide. It goes into greater technical detail about the design and implementation of GDL. It also gives detailed instructions for using the system. Appendix B consists of the header and source files for two language tokens that are added to the GDL system as an example in appendix

A. Appendix C contains the source code for a simple program which illustrates some of the advanced uses of GDL.

II. DESIGN

Any robust graphics system requires a sound, extensive design. General purpose systems, like GDL, need to be flexible, easy to understand and use, simple to maintain and efficient. A comprehensive design process is needed to ensure broad applicability. GDL was designed to be used in a broad spectrum of applications; from beginning graphics programs to the most complex projects in the Graphics and Video Laboratory. The design of GDL was shaped over an extended period drawing on numerous suggestions and requests from potential users.

A. GDL DESIGN

1. Goals and Requirements

The design of GDL seeks to satisfy many diverse goals:

1. Backward compatibility with previous NPSOFF [Ref. 5] definition files.
2. Easy extension and addition of language elements.
3. Simple maintenance.
4. Simple, easy-to-understand user interface.
5. High efficiency with respect to graphics intensive applications.

These design goals were addressed using the object-oriented paradigm and C++. The object oriented paradigm was the ideal choice for many reasons. Foremost is the paradigm's support for a naturally expressive design and the close correspondence between the design and implementation. The abstraction, encapsulation and inheritance facilities inherent in the object oriented paradigm directly supported goals 2 and 3. The C++ language was chosen for:

1. Its compatibility with existing C libraries.
2. Its efficiency.

3. The degree of control and flexibility afforded the designer/implementor over object and non-object oriented construct usage.

4. The availability of implementations on Silicon Graphics Inc. workstations.

2. Design Methodology

The design of GDL was primarily responsibility driven and atomically oriented. Because GDL is a library of classes available to the application developer, each leaf class is designed to be stand-alone in functionality. Function specific classes (e.g. a displayer) were not used because in order to approach the ideal of adding a new class to the main hierarchy without affecting any other class modules. This can't be realized in a system where knowledge of all classes is localized.

B. FUNCTIONAL OVERVIEW

1. Tokens

The GDL language is divided into atomic units called ***Tokens***. Each Token is responsible for:

1. Initializing itself to a valid default state.
2. Reading itself from an ASCII stream.
3. Writing itself to an ASCII stream.
4. Defining itself if applicable (explained below).
5. Displaying itself if applicable.

Also, a Token can be copied, compared for equality and order and may provide specific methods to access specific capabilities.

The set of Tokens is divided into five major subsets: Deftokens, Settokens, Drawtokens, Xformtokens and Othertokens. Each of these is described in the following sections.

The Token hierarchy, shown in Figure 2.1, provides the major functionality for the GDL language system. Each token contributes a specific graphical, algorithmic or managerial function. Tokens are combined to produce the desired appearance and behavior.

In order to support this functionality, several underlying systems had to be built. These systems are a run-time type identification system, an object persistence system and a simple garbage collection system. Most of these systems are provided by other language environments. However these systems and the functionality they support are an inherent part of the design. Language support for systems such as these is an important consideration and can affect design decisions. These sub-systems are described in Chapter V.

2. NPSobjects

The other hierarchy in GDL is the ***NPSobject*** hierarchy. An NPSobject is the user's normal interface to the language system. An NPSobject's basic responsibilities are similar to a Token's:

1. Read/ initialize from an ASCII stream.
2. Write to an ASCII stream.
3. Define itself.
4. Display itself

At the high level, an NPSobject represents a graphical object used in an application. These can be as simple as a single polygon or sphere or as complex as an aircraft or ground vehicle. Also NPSobjects can be used to represent objects that do not display such as an object that contains all of the lighting material definitions for an application. At the low level, an NPSobject is a collection of tokens that represents the properties and description of the NPSobject. In normal usage, an NPSobject is semantically and logically associated with an ASCII file containing token definitions. This file represents the definition of the NPSobject in the GDL. Although this is the normal usage of the GDL system, the user is not limited to this mode. The individual Token subclasses can be used directly and initialized at run-time. Likewise, NPSobjects can be created and modified at run-time independent of a definition file. This provides a flexible framework for both application developers using GDL and tool developers manipulating NPSobjects and Tokens.

3. The Grand Scheme

The GDL system is designed to be highly polymorphic and uses many aspects of the object oriented paradigm. The capabilities of the C++ language are also fully exploited.

The Token class specifies the common interface for all tokens. This allows the NPSobject class to become a client of the Token hierarchy by only needing to know how to use Token objects. Using the facilities of inheritance, an NPSobject can maintain and operate on a list of tokens in a generic manner. In other words, the NPSobject can have a list containing many different Token subclasses but can operate on each of the tokens similarly. The idea of operating on collections of heterogenous objects in a homogenous way is at the heart of the power of object oriented programing.

This capability allows the addition of new tokens to the Token hierarchy with no effect on NPSobjects. This is very important. Since the typical application will only deal with NPSobjects and NPSobjects in turn deal only with Tokens, the language designer can add new language tokens without breaking user applications. In this case the user would need to link to the new library. Otherwise, the GDL system ignores token identifiers it does not recognize.

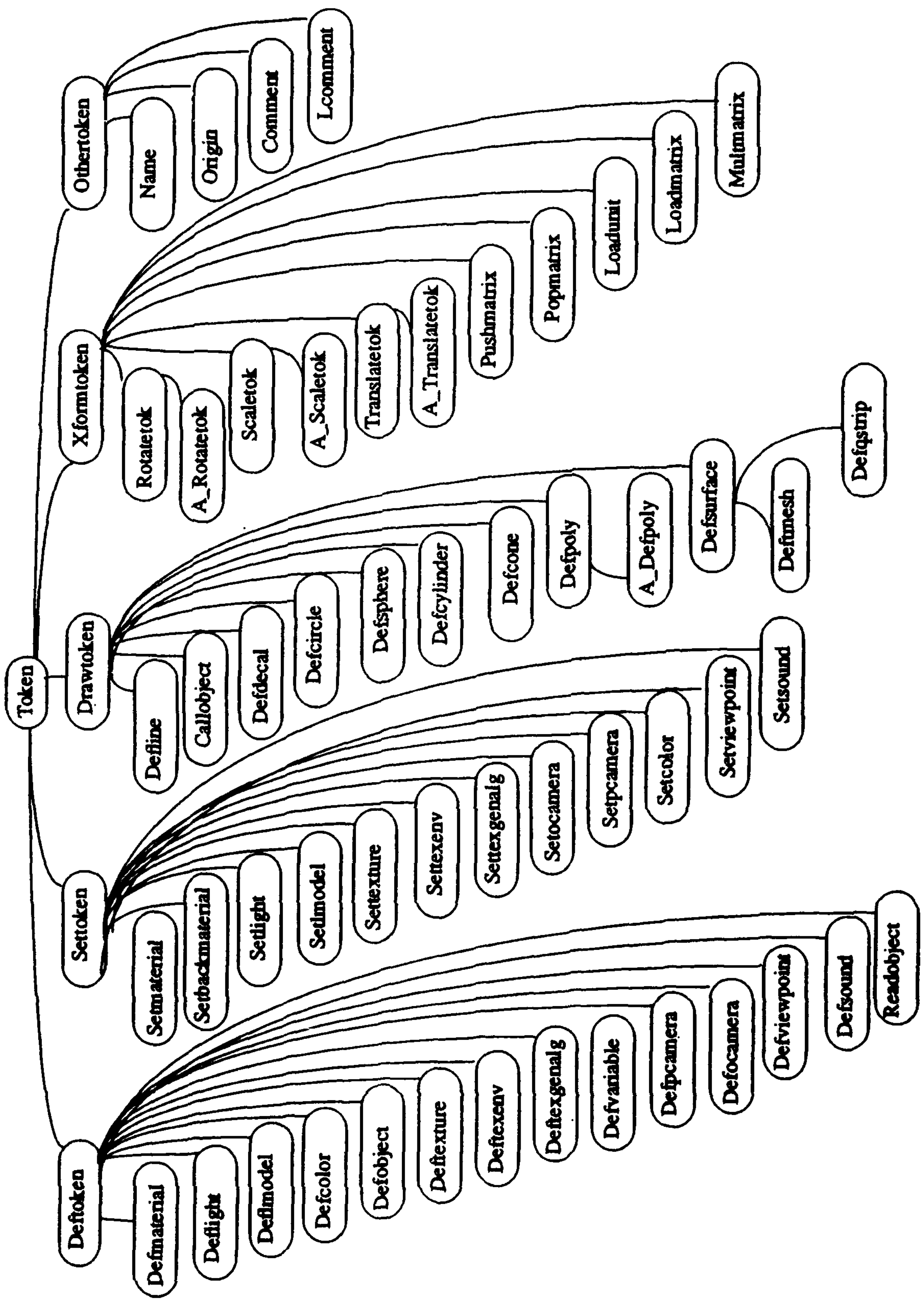

Figure 2.1 Token Class Hierarchy

III. LANGUAGE DESCRIPTION

The GDL language contains approximately 53 tokens. Each token belongs to one of five groups: Deftokens, Settokens, Drawtokens, Xformtokens and Othertokens. Each token is represented by a separate class with the token groups serving as super-classes and the Token class as the root of the hierarchy as depected in Figure 2.1. The GDL language is context-sensitive with the parsing responsibility distributed across the token hierarchy. Each language token consists of an identifier followed by a known or limited number of fields representing properties of the token and associated values. Token identifiers and field identifiers are case insensitive and the order of fields is generally not important. This provides a simple flexible syntax that is easy to remember. The user need only remember field names and valid values, not neccesarily the order they must be listed. The identifier for each token in the language is known to the root of the hierarchy and accessible by a class method that reads a stream, recognizes identifiers and creates specific token instances, which initialize themselves from the stream. When a token instance has initialized itself, the stream is positioned at the next identifier and the process continues.

The following sections describe the specific token groups. Simple examples are given for each of the tokens illustrating their formats. All values are floating point. Color values are given in the range 0.0 - 1.0. For the sake of brevity, color values are denoted by "r g b" and optionally "a" for alpha values. The term "vertex" is used to represent three component coordinate values. Unit normals are used for all normal vectors. Comments are placed after double slash (//) characters. Comments are not part of the token formats. It is important to note that many of the components of a token offer reasonable defaults and are not required.

A. DEFTOKENS

Deftokens represent graphical entities that are stored for later access and use. Some Deftokens correspond to system settings that must be defined prior to use. Others simply store information to be used later. A good example of a Deftoken is a lighting material definition which must be system defined before it can be used for rendering. Each Deftoken subclass provides a random access table which stores copies of each instance created. These tables are available to complimentary Settokens and the user. Thus each Deftoken facilitates pools of instances that can be accessed as needed. Each Deftoken has a string name that uniquely identifies it. The Deftoken class is responsible for reading and writing this name. Individual Deftoken subclasses are responsible for any fields after the name. All Deftoken formats begin with the token identifier and end with "defend". This allows the property identifiers and values to be given in any order. The current Deftoken subclasses are:

1. Defmaterial

This token represents a lighting material definition. Material definitions specify how light interacts with a surface. These definitions are made known to the graphics system and are available to the rendering process as needed. The Defmaterial token has the following format:

```
defmaterial name
 ambient r g b
 diffuse r g b
 emission r g b
 specular r g b
 shininess f //(range 0.0 - 255.0 0=no shininess)
 alpha a
defend
```

2. Deflight

This token represents a light definition. Light definitions are used during rendering to color and shade visible surfaces. Each light is predefined to the graphics system for quick access during rendering. Deflights support local and infinite light sources, colored lights and spotlights as specified in [Ref. 12]. The deflight format is:

```
deflight name
 ambient r g b
 lcolor r g b
 position x y z w  // w=0 infinite x,y,z is direction vector
                   // w=1 local x,y,z is location
 spotlight exp spreadangle
 spotdirection x y z
defend
```

3. Deflmodel

This token represents a lighting model definition. The lighting model controls how lights and materials are used and the calculations employed. Example properties of a Deflmodel are: ambient light color, whether the viewer is local, attenuation factors and whether two-sided material lighting is enabled.

```
deflmodel name
  ambient r g b
  localviewer yes|no
  attenuation K0 K1 K2 // constant, linear and squared coeff.
  twoside yes|no
defend
```

4. Defcolor

This token represents a simple rgb color used for rendering when lighting is not desired.

```
defcolor name
  r g b
```

5. Deftexture

This token represents a texture mapping definition. It associates an image file name with the texture mapping options available from the Silicon Graphics Inc. Graphics Library [Ref. 12]. The image file format is the Silicon Graphics Inc. format. The Deftexture has the following format:

```
deftexture name
```

```
  imagefile filename
  components num
  minfilter point|biliner|
        mipmap_point|mipmap_linear|
        mipmap_bilinear
  magfilter point|biliner
  wrap repeat|clamp
  wrap_s repeat|clamp
  wrap_t repeat|clamp
  tile f f f f
// wrap, wrap_S & wrap_t, and tile
// are mutually exclusive
defend
```

6. Deftexenv

This token represents environmental settings for texture mapping. It supports texture decaling, modulation and blending as specified in [Ref. 12].

```
deftexenv name
  blend|decal|modulate
  color r g b a
defend
```

7. Deftexgenalg

This token allows the user to specify parameters for the automatic generation of texture coordinates for polygons. It uses the coordinate generation capabilities available through the Silicon Graphics GL library [Ref. 12].

```
deftexgenalg name
 sdir linear|contour|spheremap a b c d
 tdir linear|contour|spheremap a b c d
// a b c d specify plane eq. coefficients
defend
```

8. Defobject

This token allows the user to group any set of tokens into a named object that is stored for use by any other object in the system. Defobjects are normally subobjects used by NPSobjects or other Defobjects. An example would be the wheel of a vehicle. One

Defobject would represent the wheel. Any object needing the wheel could reference it using the Callobject token discussed below.

```
defobject wheel
  setmaterial wheelcolor
  defpoly high // see description below
     0.0 0.0 1.0
      8
      0.0 0.0 0.0
      :
      :
      -0.1 0.1 0.0
defend
```

9. Readobject

This token gives the user the ability to access other GDL object files from an object file. The name of a Readobject token is interpreted as a filename. The file is opened and read as an NPSobject. It is assumed that the file contains Defobject definitions which are in turn stored in the Defobject table for later use.

```
readobject wheel.gdl
```

10. Defvariable

This token represents a user defined constrained floating point variable that is updated and maintained by GDL. As explained in the animation section above, Defvariables can be used in place of values in certain tokens. In this way, Defvariables offer simple animation support within GDL. The value of a Defvariable can be updated each display loop or per unit time (seconds). It can oscillate between its minimum and maximum value or wrap around. Also an update function (rule) can be named to correspond to a user provided function that updates the variable value. The Defvariable still maintains the constraints. This makes complex behaviors possible. Animated variables are discussed further in Chapter IV. The format for the Defvariable is:

```
Defvariable name
  min minval
  max maxval
  init initialval
  inc increment // per sec if timed
  timed yes|no
  wrap yes|no
  rulename rname
defend
```

11. Defpcamera

This token allows the user to specify a perspective viewing frustum. The Defpcamera can then be used in conjunction with a Defviewpoint to define the view volume used for rendering. The field of view for a Defpcamera can be specified by a fixed value or a Defvariable. Thus the field of view can be animated to support effects like zooming.

```
Defpcamera name
  nearplane nz
  farplane fz
  aspect aratio
  fov angle|Defvariable name
defend
```

12. Defocamera

This token is similar to the Defpcamera. It supports orthographic projections. The user specifies the view volume by setting the clipping planes in each dimension.

```
Defocamera name
  xdim xmin xmax
  ydim ymin ymax
  zdim zmin zmax
defend
```

13. Defviewpoint

This token is used to specify the orientation of a camera. The parameters are the location of the viewpoint, the location of the reference point and the twist angle of the view volume. Each location can be specified using a combination of fixed values and Defvariable names as well as a single Defobject name. The twist angle can also be specified by a Defvariable name for continuous update. The flexible format for specifying locations and the animation support through Defvariables make Defviewpoints interesting and easy to use.

```
Defviewpoint name
  from x|Defvariable name y|Defvariable name z|Defvariable name
      |Defobject name
  to x|Defvariable name y|Defvariable name z|Defvariable name
      |Defobject name
  twist angle|Defvariable name
defend
```

B. SETTOKENS

Settokens represent entities that change the graphics state. They are normally paired with a Deftoken and provide for the use of the Deftoken's properties. Thus the Deftoken and Settoken classes are cooperative with the Deftokens containing a pool of definitions and the Settokens providing access to the definitions. Most Settokens have the property that only one (of each type) is active at any time. Each Settoken has a name. The Settoken class is responsible for reading and writing this name. Also a Settoken is usually specified as on or off. Most Settoken formats use only a single line of text so there is no end marker as with Deftokens. The different Settokens are described below.

1. Setmaterial

This token enables the use of a material definition. It makes the named material current for use during rendering assuming the name corresponds to a valid Defmaterial. The format for a Setmaterial is:

```
setmaterial name
```

2. Setbackmaterial

This token is the same as the Setmaterial token except it specifies which material to use for the backs of polygons when two-sided lighting is enabled.

```
setbackmaterial name
```

3. Setlight

This token allows the use of lighting definitions. A light may be on or off. Up to eight lights may be on at one time. The format for Setlight is:

```
setlight name num on|off
```

4. **Setlmodel**

This token activates a named lighting model. Setlmodels can be on or off.

```
setlmodel name on|off
```

5. **Setcolor**

This token sets the current drawing color to a previously named and defined color.

```
setcolor name
```

6. **Settexture**

This token enables texturing using a previous texture definition. A Settexture can be turned on or off to allow texturing of portions of an object.

```
settexture name on|off
```

7. **Settexenv**

This token selects a texturing environment definition for use.

```
settexenv name on|off
```

8. **Settexgenalg**

This token selects the algorithm definition to use for automatic texture coordinate generation. Also the user can specify whether coordinates should be generated along the s axis or t axis or both.

```
settexgenalg name sdir on|off tdir on|off
```

9. **Setpcamera**

This token selects a Defpcamera definition for use. The Setpcamera can be on or off. This enables interactive control of cameras and the switching of view volume specifications between user code and GDL tokens.

```
setpcamera name on|off
```

10. **Setocamera**

This token is the same as the Setpcamera but for orthographic cameras.

```
setocamera name on|off
```

11. Setviewpoint

This token allows the user to select a previously defined viewpoint for rendering. The Setviewpoint token is used in conjunction with the Setpcamera and Setocamera tokens to define the viewing volume location and orientation.

```
setviewpoint name on|off
```

C. DRAWTOKENS

Drawtokens represent the visible graphical entities. These are the items in a scene that all other tokens affect. Most of the Drawtokens represent low-level graphical constructs but several offer higher level support. All Drawtokens have a resolution value. The Drawtoken class is responsible for reading and writing the resolution. The resolution can be high, medium or low. The Drawtoken resolutions are used to support multi-resolution objects in applications. A description of the Drawtokens in GDL follows. Here *res* denotes high|medium|low resolution choices as discussed above. Also *vertex* denotes an xyz coordinate.

1. Defpoly

This token represents a basic polygon. The polygon has a single unit normal and can have texture coordinates optionally specified. The format for a Defpoly is:

```
defpoly res
  nx ny nz // normal
  numvertices
  vertex1 [s1 t1]
  :
  vertexn [sn tn]
```

2. A_defpoly

This token is the same as the Defpoly except that it supports the use of Defvariable names in place of vertex coordinate values. The "A_" prefix denotes animated behavior. The ability to specify polygon vertex coordinates that change automatically is powerful and fun.

```
a_defpoly res
  nx ny nz
  numvertices
  vertex1|defvariable name [s1 t1]
  :
  :
  vertexn|defvariable name [sn tn]
```

3. Defsurface

This token represents a planar polygonal surface with vertex unit normals. Vertex normals are used for Gouraud shading. Also texture coordinates can be optionally specified for each vertex. The following tokens, Deftmesh and Defqstrip have the same characteristics as Defsurface bu use different algorithms for rendering the surface. The format for a Defsurface is:

```
defsurface res
  numvertices
  vertex1 normal1 [s1 t1]
  :
  vertexn normaln [sn tn]
```

4. Deftmesh

This token represents a polygonal surface using a triangular mesh. It uses the same basic format as the Defsurface.

```
deftmesh res
  numvertices
  vertex1 normal1 [s1 t1]
  :
  vertexn normaln [sn tn]
```

5. Defqstrip

This token represents a polygonal surface using quadrilateral strips. It uses the same basic format as a Defsurface.

```
defqstrip res
  numvertices
  vertex1 normal1 [s1 t1]
  :
  vertexn normaln [sn tn]
```

6. Callobject

This token allows the user to access and display Defobjects. A Callobject is used to display subobjects that are defined and named separately. A Callobject token has two resolutions associated with it. One for itself pertaining to which resolutions the subobject will be shown and another to designate the display resolution for the subobject. The format for a Callobject is:

```
callobject res objname display-res
```

7. Defcircle

This token represents a simple 2D circle defined in the x-y plane. The properties of a Defcircle are its center location and radius.

```
defcircle res
  center x y z
  radius r
endcircle
```

8. Defsphere

This token allows the user to define a sphere parametrically. In addition to center and radius, the user can specify the number of longitudinal and latitudinal divisions to use to tessalate the sphere. The square root of the number of panels is used to determine the tessalation divisions. The Defsphere format is:

```
defsphere res
  center x y z
  radius r
  panels numpanels
 endsphere
```

9. Defcylinder

This token allows the user to parametrically specify a cylinder for display. Its format is similar to Defsphere with the addition of a height property.

```
defcylinder res
  base x y z
  radius r
  height h
  panels numpanels
endcylinder
```

10. Defcone

This token allows the user to parametrically define a cone for display. Its format is similar to the Defcylinder format.

```
defcone res
  base x y z
  radius r
  height h
  panels numpanels
endcone
```

11. Defline

This token allows the user to specify a multi-point line in three dimensions.

```
defline res
  numpoints
  vertex_1
  :
  :
  vertex_n
```

12. Defdecal

This token provides the user with a facility to define decaled polygons. Decaling is a technique for properly rendering co-planar polygons while using z-buffered hidden surface elimination. With the Defdecal token, users can specify the components of the underlay and overlay portions of a decal. The components of the underlay and overlay can be any displayable token (Drawtoken, Settoken and Xformtoken). Drawtokens must be planar. In normal use, only Setmaterial and Defpoly tokens are used.

```
defdecal res
  underlay
   defpoly1 // see format above
   defpoly2
   :
  overlay
   defpoly1
   defpoly2
   :
defend
```

D. XFORMTOKENS

Xformtokens represent entities that alter the normal representation of Drawtokens at display time. Xformtokens represent run-time manipulations and do not alter the definition of other tokens. The normal transformations of rotation, translation and scaling are represented. Also there are tokens that relate directly to capabilities of the Silicon Graphics hardware and rendering process. Following is a brief description of each of the Xformtokens.

1. Rotate

This token represents a single axis rotation. The rotation is normally performed in world space with the rotation specified by a float representing whole and fractional angles. The format for a Rotatetok is:

```
rotate x|y|z angle
```

2. A_Rotate

This token is the animated version of a Rotatetok. It accepts a Defvariable name for the rotation value. This token is *useful for simple animations* like wheels or propellers turning. It allows the developer to specify the animation constraints and leave the rest to the GDL system.

```
a_rotate x|y|z angle|name
```

3. Translate

This token represents a simple translation in 3-space.The format is:

```
translate dx dy dz
```

4. A_Translate

This token is the animated version of the Translate token. Any of the three translation values can be linked to a Defvariable for automatic update.

```
a_translate dx|name dy|name dz|name
```

5. Scale

This token represents a 3D scale.

```
scale sx sy sz
```

6. A_Scale

This token allows the use of Defvariables for animated scaling.

```
a_scale sx|name sy|name sz|name
```

7. Pushmatrix

This token gives the user access to the pushmatrix function in the SGI GL [Ref. 12] library. This function saves the state of the rendering transformation matrix. Using Pushmatrix and Popmatrix tokens, the user can specify hierarchical transformations.

```
pushmatrix
```

8. Popmatrix

This token gives the user access to the popmatrix function in the SGI GL [Ref. 12] library. The popmatrix function removes the current transformation matrix from the matrix stack.

```
popmatrix
```

9. Loadmatrix

This token takes a user defined 4x4 matrix and initializes the hardware matrix stack with it.

```
loadmatrix
  a b c d
  e f g h
  i j k l
  m n o p
```

10. Multmatrix

This token allows the user to multiply a user-defined 4x4 matrix onto the matrix stack.

```
multmatrix
  a b c d
  e f g h
  i j k l
  m n o p
```

11. Loadunit

This token allows the user to initialize the transformation matrix stack with unit matrix.

```
loadunit
```

E. OTHERTOKENS

Othertokens are auxiliary tokens that do not directly affect the graphical representation of an object. They are not related except for this fact. The current Othertokens are described below:

1. Name

This token allows the user to name an NPSobject. When a GDL definition file is read by an NPSobject, the object scans for Name tokens and saves the last one for user inquiries. This is mostly for documentation use.

```
name objname
```

2. Origin

This token specifies the origin or reference point for an NPSobject. Like the Name token, an NPSobject scans the definition file for Origin tokens. At run time, the user can query an NPSobject for its origin. This information is useful for transformations and viewing.

```
origin 0 0 0
```

3. Comment and Lcomment

These tokens facilitate C++ style comments in GDL definition files. The Comment token is delimited by the "/*" "*/" pair and can be multi line. The Lcomment is a single line comment and is delimited by "//".

Overall, the GDL language is fairly simple and easy to remember and understand. Yet it offers a great deal of flexibility and power through higher level support for graphical abstractions. An example GDL object description file is shown in Figure 3.1, Figure 3.2 and

Figure 3.3. Here a simple model of a planet and moon is described. The planet and moon are textured spheres and both rotate under GDL control, both about their axis and in orbit in the case of the moon. The example illustrates the use of many of the language tokens and simple animation. A picture of twoframes of the scene that this file describes is shown in Figure 3.4 and Figure 3.5.

```
name planets
origin 0 0 0

/* This gdl file represents an earth-like planet and a single moon
   The two planets are textured and are rotated using animated
   variables.
*/

// define the Sun
deflight sun
  ambient 0 0 0
  lcolor 1 1 0.75
  position -1 0 0.25 0
defend

// define some materials

defmaterial sky_blue
 emission 0.0 0.0 0.0
 ambient 0.105882 0.161569 0.200000
 diffuse 0.529412 0.807843 1.000000
 specular 0.0 0.0 0.0
 shininess 0.0
 alpha 1.0
defend

defmaterial aquamarine
 emission 0.0 0.0 0.0
 ambient 0.099608 0.200000 0.166275
 diffuse 0.498039 1.000000 0.831373
 specular 0.0 0.0 0.0
 shininess 0.0
 alpha 1.0
defend

defmaterial gray4
 emission 0.0 0.0 0.0
 ambient 0.007843 0.007843 0.007843
 diffuse 0.039216 0.039216 0.039216
 specular 0.0 0.0 0.0
 shininess 0.0
 alpha 1.0
defend

// define the textures
deftexture earth
 imagefile earthclouds.rgb
 minfilter mipmap_bilinear
 magfilter bilinear
 wrap repeat
endtexture

deftexture moon
 imagefile moon.rgb
 minfilter mipmap_bilinear
 magfilter bilinear
 wrap repeat
endtexture
```

Figure 3.1 Example GDL object file

```
deftexenv planetenv
  modulate
defend

deftexgenalg earthalg
  sdir linear 0.0 0.1 0.5 20.0
  tdir linear -0.10 0.0 0.1 20.0
defend

deftexgenalg moonalg
  sdir linear 0.0 0.1 0.1 0.0
  tdir linear -0.1 0.0 0.1 0.0
defend

// the animated variables for rotating the planets
defvariable earthrot
  min 0
  max 360
  init 0
  inc .1
  timed yes
  wrap yes
defend

defvariable moonorbit
  min 0
  max 360
  init 0
  inc 0.15
  timed yes
  wrap yes
defend

defvariable moonrot
  min 0
  max 360
  init 0
  inc 0.2
  timed yes
  wrap yes
defend

// this is the displayable portion
setlight sun 2 on
settexenv planetenv on

// save the state of the application transformation matrix
pushmatrix

// incline the scene
rotate z 23.5

pushmatrix
// spin the earth continously
a_rotate y earthrot

// color and texture the earth
setmaterial sky_blue
settexgenalg earthalg sdir on tdir on
settexture earth on
```

Figure 3.2 Example GDL object file (cont.)

```
// draw the earth, make it low res visible
defsphere low
  center 0 0 0
  radius 15
  panels 200
endsphere
settexture earth off

// draw the axis
setmaterial gray4
defcylinder med
  center 0 -20 0
  radius 0.5
  panels 10
  height 40
endcylinder

popmatrix

// isolate the moons motion
pushmatrix

// continously rotate the moon about the earth
a_rotate y moonorbit
translate -40 0 0

// isolate the moons spin
pushmatrix

a_rotate y moonrot

// color and texture the moon
setmaterial aquamarine
settexgenalg moonalg sdir on tdir on
settexture moon on

defsphere high
  center 0 0 0
  radius 5
  panels 100
endsphere
settexture moon off

// use a predefined material for an object on the moon
setmaterial brass
rotate z -90
defcone medium
  center 0 0 0
  radius 1.5
  height 7
  panels 10
endcone

// recover the state of the transformation matrix
popmatrix
popmatrix
popmatrix
```

Figure 3.3 Example GDL object file (cont.)

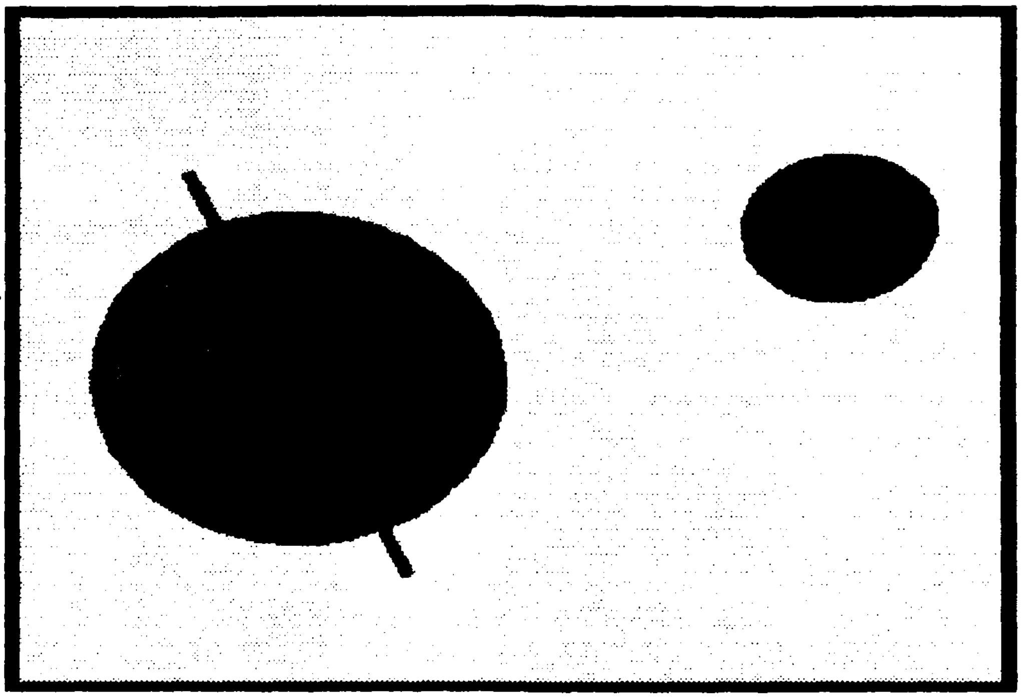

Figure 3.4 Sample display of animated GDL object

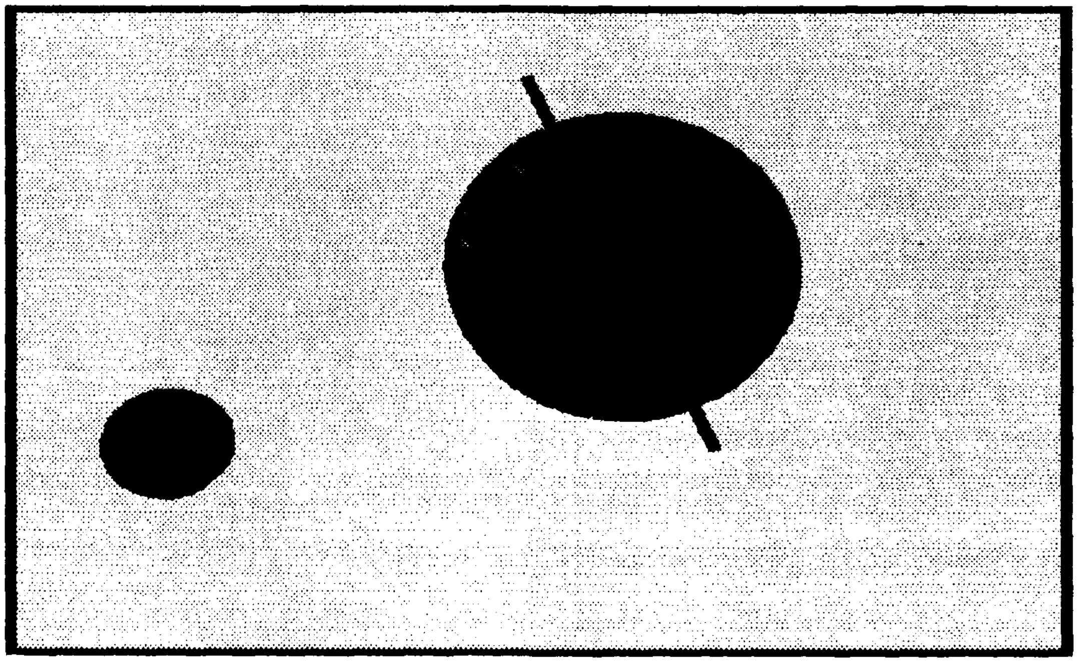

Figure 3.5 Sample display of animated GDL object (cont.)

IV. ANIMATION AND MULTI-RESOLUTION DISPLAY

Two of the important capabilities that GDL supports are simple animation and multi-resolution display. These abilities allow the application designer to easily manage and represent complex objects by letting GDL handle lower level details.

A. ANIMATION CONTROL

Animation is supported through user defined variables that are automatically updated by the GDL system. These animated variables are similar to the animated basic types described in [Ref. 11]. The user can place simple constraints on the variable and let the GDL sysytem update it linearly or a user defined rule can be specified for the system to use for variable updating.

An animated variable can be used by a variety of tokens in place of fixed values. The animated variable token, Defvariable, acts as the server and other tokens as clients. The client token sets up an association with the variable and informs the variable to update itself prior to each display cycle. This scheme supports a wide range of simple, continuous animations. For example, if a developer wanted to model a lighthouse with a rotating beacon, the rotation could be controlled by an animated variable using constraints defined by the developer. The beacon rotation would then be updated entirely by GDL leaving the developer to concentrate on higher level details. A simple example illustrating this is shown in Figure 4.1.

```
// we'll assume the existence of predefined objects that we can
// call from this object description. The tokens used here are
// described later.

Defvariable lightrotater
  min 0
  max 360
  init 0
  inc 18.0
  timed yes
  wrap yes
defend
// This defines a variable that linearly varies from 0.0 to 360.0
// at 3 rpm (18 deg / sec) wrapping from 360 to 0.

// make the object high res and
// display the lighthouse at high resolution
 callobject high lighthouse high
// display the rotating beacon (note transformations are isolated)
pushmatrix
a_rotate y lightrotater
callobject high lighthouse_beacon high
popmatrix
```

Figure 4.1 Simple Animation Example

If more complex behavior is needed by the animated variable, the developer can specify a rule (function) for updating the variable. The name of the rule is given and the variable associates to the rule at run time from a table initialized by the user. This rule is then used to update the animated variable's value while the GDL system maintains the variable constraints. An example of this is shown in Figure 4.2. An important point to remember when using animated variables is that they are updated by their clients each display cycle. For example if two A_Rotate tokens use the same Defvariable to animate the rotation, each will tell the Defvariable to update when it is displayed. Thus the animated variable will be updated twice during a single cycle. This is an simple but potentially important consequence of using a single Defvariable in multiple animated tokens.

```
// In object description file: ruletester.gdl
Defvariable rulevariable
  min -100
  max 100
  init -100
  rule accel_rule
defend

// defvariable would be used where needed in animated
// tokens

/************************** In User Program ********/

// define the rule function. It takes the last value
// and scales it.
float accel_rule(float val) {return val * 1.05;}

// inside main() we add the rule to a table for GDL
// access

main() {  // first init the ruletable with our rule
  Defvariable::ruletable->add("accel_rule",accel_rule);

  NPSobject uses_rule("ruletester.gdl");

  // now we have an object that uses the rule
  // set up graphics environment and open a window here

  uses_rule.define(); // define the object

  uses_rule.display(); // This would be in the display
                       // loop. The Defvariable would
                       // call the rule function each
                       // time the object is displayed
                       // to update the value of the
                       // variable
}
```

Figure 4.2 Animation example using a rule

B. MULTI-RESOLUTION DISPLAY

Multi-resolution display is important to complex virtual world applications. Our experience has been that most of our applications are graphics bound. Much of our effort goes to reducing the number of primitives sent to the renderer. GDL supports three drawing resolutions for each of its Drawtokens: high, medium and low. GDL supports both a simple and a hierarchical resolution scheme. An NPSobject can be displayed at any of the three resolutions under user control. Single level mode displays only the tokens having a particular resolution. In hierarchical display mode, high resolution displays all Drawtokens, medium resolution displays medium and low resolution Drawtokens and low resolution displays only low resolution tokens. An object designer can designate which polygons, subobjects etc. are visible at each resolution. The application developer can then display NPSobjects at varying resolutions; based on distance from the viewer for instance.

This multi-resolution model supports two methods of object design. The designer can specify three distinct representations of an object or the designer can design using one set of surfaces but designate which are visible at various resolutions. The latter is much more difficult but results in smaller object sizes. Using this multi-resolution scheme, substantial performance improvements in display times are possible. The primary cost is development time of the object models.

V. IMPLEMENTATION

Many issues came up during implementation related directly to the use of C++ as the implementation language. Because of the lack of a language standard, no built-in support for run-time type information, no garbage collection and the lack of standard data structure classes, a great deal of time was spent developing these basic systems and structures before implementing GDL.

The first effort was to construct a library of standard data structure abstract data types and concrete data types. Initially several popular public domain libraries were considered including the National Institutes of Health Class Library (NIHCL)[Ref. 13] and the Texas Instruments, Inc. Library (COOL)[Ref. 14]. Although the implementation of both libraries was very educational, neither of these libraries was used for several reasons:

1. The NIHCL single root hierarchy was deemed inappropriate.
2. Designing classes to be used by the libraries was cumbersome.
3. The time investment to become proficient using the libraries was too high.
4. Difficulty in getting a completely built version of the libraries discouraged further use.

There were many good points to both libraries. These were incorporated into what was to become the Naval Postgraduate School Class Library (NPSCL). NPSCL is a collection of stand-alone classes. The classes are either concrete data types such as string, date and time, generic abstract data types for containers (lists, tables, trees) or cooperative classes comprising a support system. At the time that NPSCL was implemented, most C++ compilers did not support templates as defined in the proposed standard and [Ref. 15]. In order to implement generic templated containers, NPSCL uses macro substitution. This is not the most desirable solution but does not require special preprocessor support and is fairly easy to use. As C++ compilers supporting the C++ template facility become available, NPSCL wil be updated to use true templates.

A. RUN-TIME TYPE INFORMATION

NPSCL provides several support systems that GDL uses extensively. The first is a simple run-time typing system. Run-time type information (rtti) was not originally a part of C++ due to the added overhead such a system would impose. A recent proposal [Ref. 16] to add rtti has been put forth by the language's designer which should correct a major deficiency. Many C++ applications do not need run-time typing support relying instead on virtual methods and dynamic binding of method calls. This falls apart in systems like GDL. All tokens behave similarly and respond to the same messages and can thus be managed generically as tokens. However, there are many occasions when some specific behavior not common to all tokens is needed from a token. Also there are occasions when certain tokens must be separated from the rest. Both of these situations demand a consistent way to identify the type of object referenced and safely cast pointers down the inheritance hierarchy. This is the purpose of the run-time type system of NPSCL. The type system is non-intrusive meaning that not all classes must participate, although there is little reason not to include all classes. The system is based on that described in [Ref. 17] and similar to the system described in [Ref. 16]. Basically each class has a public static data member that contains a string identifier for the class and a list of immediate base classes for the class. Methods and macro support allow the user to query the type of a class, determine if a downcast is safe, compare for type equality and other helpful functions. This system is useful and effective. It imposes little space overhead on client classes and very little performance overhead by using inline methods where possible. Also since run-time type inquiries are the exception rather than the norm, the system did not introduce significant performance penalties in the tests conducted. In addition to the string name based type system described, NPSCL provides a simple object identification capability based on integers. Using this system, each class instance is given a unique integer identity that can be used for more refined identity testing. These integer identifiers are used by GDL for various Silicon Graphics, Inc. GL functions [Ref. 12] among other things.

B. REFERENCE COUNTING

Another system extensively used by GDL is a simple reference counting garbage collection system. In order to save space and improve efficiency, many token objects are shared between each other and NPSobjects. For example, consider two NPSobjects. Each is associated with a description file that defines a "gold" Defmaterial. Only one copy of the Defmaterial is wanted but each NPSobject that defines the same Defmaterial must contain a copy in the event that the NPSobject must write itself to a file. In this case, the first Defmaterial read would allocate space and insert a pointer in a table. The second NPSobject would get a pointer to the first instance rather than a new one. To support this with a minimum of developer worry, NPSCL provides a simple reference counting system using smart pointer objects. This system is based on the examples in [Ref. 18] and are similar to the "letter-envelope idiom" in [Ref. 19]. Basically every class in the GDL system contains a reference count member and methods to increment and decrement the count. If an object's reference count decrements to zero then its memory is returned to the memory system. The management of the reference counts is the responsibility of a friend class that encapsulates a pointer to the referenced class. This class is generic and uses templates to provide type safety. The pointer class, called a Refptr in NPSCL, overloads operators to behave as a normal pointer with the addition of adjusting reference counts as pointers are assigned, copied and destroyed.

There are several advantages and disadvantages to using this system. Many relating to garbage collection in general:

Advantages:

1. System is simple and easy to use.
2. Frees developer from many memory management chores.
3. System is non-intrusive. It can be used or not used as desired.

Disadvantages:

1. System imposes overhead on pointer manipulation. Little overhead is imposed for pointer use or access.

2. User can break system by mixing real pointers and Refptrs.

3. System doesn't detect circular references that might result in unrecoverable memory.

Despite these disadvantages, this system is used in GDL with few problems. In normal use, the circular reference problem is not encountered and the overhead is only noticed at non-critical times such as object creation/initialization. Other phases of an object's use normally involve accessing the object pointed at and this operation has little to no overhead due to the use of inline methods. The primary limitation to using this garbage collection system is placed on the developer. The developer must ensure that real pointers are not mixed with the smart pointers across scopes. This is to prevent the system from deallocating an object still referenced by a real pointer. This is not a problem for the typical user as all object management is taken care of within GDL. It is a consideration for developers of tools and those managing custom collections of GDL tokens.

C. THE PERSISTENCE MODEL

A primary requirement for GDL is the ability to store and retrieve object definitions to secondary storage. Since C++ does not provide a standard persistence mechanism, one was designed into GDL. The persistence system was modeled after several different systems, in particular the "virtual constructor" and exemplar methods outlined in [Ref. 19]. The primary responsibility for storage and retrieval is distributed among the GDL token classes. The most derived class controls most of the process. Each token implements three methods, the read_from, store_on and creator methods as well as a special "reader" constructor called with an input stream. Each takes as input either an input or output stream. The read_from method expects the stream pointer to be located immediately after the identifier of the token and reads all fields on the stream applicable to itself until either an ending flag is encountered, as in Deftokens, or a certain number of lines have been read, as with Defpolys. The store_on method stores the token identifier and data values in the correct format on a specified stream. This method has a parameter that tells the token whether it should output its identifier with its data. Using this parameter, derived classes can have super classes output their data without inserting extraneous identifiers on the stream. The creator method is a static method. This is important because it does not work on a per object basis. The creator method's function is to allocate a new instance of a token and initialize it from an input stream. The creator then returns a pointer to the new token to the caller. The creator method acts as the virtual constructor as explained below.

Using these methods, each token provides facilities to read, write and initialize itself from file streams. Still there must be some object or process in overall control of all this. All of the token identifiers currently valid in the system must be known in order to recognize them on an input stream. Also, once an identifier is recognized, there must be a way of telling the correct token to initialize itself from the stream. This responsibility is delegated to the Token base class. The Token class contains a static table of Tokeninfo objects called the tokentable. A Tokeninfo object associates a token's typename and the address of its creator method. The Token class provides a public static method called read_token() that will process an input stream using the following general algorithm:

1. Read a string, assume it's a token identifier.
2. Lookup the identifier in the tokentable.
3. If the identifier is valid then get the tokeninfo object for that token, else issue an error and return.
4. Call the identified tokens creator method passing the input stream in.
5. The creator will return a pointer to a valid token initialized from the stream. Return the token pointer to the caller.

The read_token method is called repeatedly by the NPSobject::read_from method until the end of file is reached.

An important consideration in the design of GDL was the initialization of the tokentable. One of the primary design goals was to be able to add new tokens to the language system with minimum effect on other modules. The ideal being providing a header (.h) and implementation (.C) file for the new token(s) and adding the object module to the library archive. This ideal is very close to being met. The tokentable is initialized dynamically using a special constructor in each token and a special instantiation of each token.

The special constructor is one that takes as its sole argument a Tokeninfo object. As discussed above, the Tokeninfo object is the object placed into the tokentable. The base class Token's special constructor places the Tokeninfo object into the table. All derived tokens simple call their base class's special constructor passing the Tokeninfo object along to the root of the hierarchy.

In order for this system to work, this constructor must be used. Thus in each tokens implementation file, a single static object of the particular token type is instantiated using the special constructor. The C++ language guarantees that global static objects will be constructed before main() is entered. So all of the static objects used for tokentable initialization are constructed prior to main() ensuring that the tokentable is properly initialized automatically at run-time. There is no need for user initialization.

This system for dynamic initialization works well. There is one problem though. Current linker technology does not support this model well. Typical GDL users use only the NPSobject class and don't directly refer to the token subclasses. Since the NPSobject classes deals with tokens in the abstract through Token pointers, it does not refer to derived tokens directly either. Thus there are normally no references to derived token modules for the linker to resolve. The result is that the linker does not include the modules for derived classes and the tokentable is not initialized properly.

The temporary remedy was to fall back to the more traditional method of having an object or module that ensures that each module is linked in. Instead of having an external class manage the tokentable initialization, there is an external object, called a Token_Registrar, whose constructor calls a static method named register_token() provided by each token class. This method does nothing. It is used solely to generate an unresolved reference for the linker. The Token_Registrar is instantiated in the Token module which is always linked in. The advantage to this approach is that the registration functionality is easily removed without affecting anything.

The long term remedy is more intelligent linker technology. Dynamic linking to the degree needed is not supported on the platforms we use. Shared libraries don't address the problem either and are very difficult to construct for a system such as GDL. As object oriented programming and C++ become more popular, linker technology must improve to support the highly dynamic designs possible like this initialization scheme.

There is another consequence to using the registration method. NPSobjects only know about the class Token. The actual derived tokens used are instantiated at run-time and accessed via dynamically bound calls. Thus there is no way to know which tokens will be needed by any set of definition files. Therefore all token modules must be linked to the application. This results in large applications with possibly a lot of unused code. The alternative is to preprocess definition files and only link the modules needed. However, this is limiting and establishes an application dependence. The definition files can not be modified without preprocessing again. This option is discussed in APPENDIX A: NPSGDL USER AND DEVELOPER'S GUIDE.

D. USING GDL

The normal usage of GDL is straightforward and simple. The NPSobject class is the normal interface to the system. The user simply instantiates an NPSobject with a definition file name, defines the NPSobject to ensure that any Deftokens are defined to the graphics system and displays the object when desired. This technique is illustrated in Figure 5.1

```
// program display_planets.C
#include "NPSobject.H"
#include "gl.h"
void init()
{
  // open and config a gl window
  winopen("planets");
  RGBmode();
  doublebuffer();
  gconfig();
}

main() {
float backcolor[4] = {0.0, 0.0, 0.0,
0.0};
NPSobject p_obj("planets.gdl"); // read the file


init();
p_obj.define(); // define any Deftokens

while(1) { // kill from win manager
  c4f(backcolor);
  clear();  // clear the window
  // put up viewing and transformation stuff

  p_obj.display();
  swapbuffers();
  }
}
```

Figure 5.1 GDL object file usage example

In addition to this object definition-file based approach, the user can instantiate and use individual tokens. This way the GDL tokens can be used as individual abstractions of the underlying graphics library. A simple example of this is shown in Figure 5.2. The key point is that normal use of GDL is simple since management of low level details is hidden from the user. As mentioned before, GDL provides a standard method of describing graphical objects. Since a large number of objects exist in our laboratory, few developers need to worry about the details of creating description files. Many simply use what is available thereby speeding the prototyping and development process.

```
// program use materials.C
#include "NPSobject.H"
#include "Setmaterial.H"

void init()
{
  // open and config a gl window
  winopen("planets");
  RGBmode();
  doublebuffer();
  gconfig();
}
main() {
float backcolor[4] = {0.0, 0.0, 0.0,
0.0};

// read the file containing material
definitions
NPSobject m_obj("allmaterials.off");
init();
m_obj.define(); // define all materials


while(1) { // kill from win manager
  c4f(backcolor);
  clear();  // clear the window
  // put up viewing and transformation stuff

  Setmaterial cur_mat("gold");
  // draw something that is gold
  swapbuffers();
  }
}
```

Figure 5.2 GDL token usage example

VI. PERFORMANCE CONSIDERATIONS

GDL is used in complex, interactive, real-time 3D graphics applications. Performance is a critical issue. Limited analysis has shown that GDL is very efficient when compared to previous systems with similar purposes [Ref. 5]. GDL does impose a small amount of overhead in managing objects manually using traditional methods. This is primarily due to the trade-offs between designing for application use and general tool use. Many of the data structure classes used support high level operations that will be helpful to GDL tool designers. While this does not imply that the data structures are inefficient, it does increase the code size. A good example is the use of a string class that manages memory and supports substrings and concatenation compared to normal C character arrays. But as the objects become complex, the overhead of GDL factors out. Also the flexibility and utility of the system compensates for any minor performance degradations.

GDL performance is based on two main strategies. First, all calculations are preprocessed prior to display. Second, routines that are called often or are demonstrated bottlenecks are optimized based on profiler feedback.

The first strategy has the most impact. Where possible, each token performs any calculations needed either during initialization or definition. Also graphics data is cached for fast retrieval rather than calculated during display. This is the classic trade of space for speed. For example, during initialization, the Defsphere token calculates the vertices for a tessellated sphere and stores the results in an array for easy access. This technique is not new but important and applicable.

Another performance boost is gained in the NPSobject class. Each NPSobject contains five lists of Tokens. The main list contains all tokens and is the basis for the other lists. The main list is used for reading and writing. The second list contains only Deftokens, while the other lists are used for display and contain tokens of the same resolution. This approach reduces the number of function calls required at different phases of object use. Remember that each token responds to a common set of messages. If the token does not directly handle a message, for instance a Deftoken does not implement a display() method, then the default method is invoked in a base class, typically Token. This default method is usually empty but still imposes a function call. These unneccesary calls are prevented by using specific lists at the expense of space and preprocessing time. Also the separate lists, coupled with the multi-resolution display model, reduces the number of primitives sent to the underlying graphics system.

The second strategy involves the normal performance tuning cycle. Profilers were used to identify those functions that were called the most and those that used the highest percentage of CPU time. These were then analyzed for optimization.

The primary method during the first optimization phase was the use of inline functions. This came into play in two ways. The first way is the normal use of the C++ inline construct. During design, only the most simple methods were designated as inline, reserving it's use for tuning. This was to prevent code size growing to the point of generating excess page faults in the virtual memory system. This proved prudent as additional inlining was not needed very much. Functions that were called fairly often that were candidates for inlining were made inline. The page fault rate was monitored for excess increases as a result of the inlining.

The second inlining method used was to reduce or eliminate member function calls from within a class. While the function calls make the code easier to read and more compact, they result in one or more function calls in order to accomplish something with data already accessible to the first member function. So, where feasible, each member function manipulates class data explicitly rather than through other member functions. This is also true in the case where the other member function is inline. Since the inline directive can be ignored by the compiler, the inline expansion was used directly when needed.

After analyzing critical functions, some general and machine specific optimizations were made. [Ref. 20] was used as a guideline for optimal use of the graphics architecture. One of the first optimizations was to unroll loops during display. For example, a Defpoly contains a list of n vertices. If n is small, less than seven, the traversal loop is unrolled yielding better performance. Also pointer arithmetic is used over subscripting when it is safe and convenient to do so.

One of the machine specific techniques used is to use four element float arrays for vertices rather than three elements. This is due to the read length of the cache system in Silicon Graphics 4D series workstations.

GDL drawing performance was compared to that of NPSOFF [Ref. 5], a system with similar behavior implemented in C. The comparison consisted of displaying the same object 10000 times and reporting the average user and system time to display a single frame. The object displayed was a Gouraud shaded, lit cube, with two textured sides, a small line and a simple decal of a triangle on a square.The texturing was done with both explicit coordinates and automatically generated texture coordinates. The object was rendered on a Silicon Graphics, Inc. Iris 4D/340 VGX using a perspective projection and z-buffered hidden surface removal. The results of five runs of each timing program are shown below with user time and system time in microseconds:

Table 1: GDL and NPSOFF Performance Comparison

SYSTEM		Run 1	Run 2	Run 3	Run 4	Run 5	Avg.
NPSOFF	User Time	1438	1491	1501	1645	1629	1541
	System Time	900	828	923	870	919	888
GDL	User Time	1466	1371	1432	1470	1572	1462
	System Time	874	932	924	1033	894	931
Total Time	NPSOFF 2429	GDL 2393	1.4% Diff.				

These results are encouraging. While the two systems share common capabilities, GDL provides many additions and improvements over NPSOFF. In addition to the benefits gained from the object-oriented design and implementation, GDL provides extensive error detection and recovery, reasonable default behavior, extended data structure support, name space control and more high level application support.

VII. LIMITATIONS, FUTURE WORK AND CONCLUSIONS

GDL is the most recent effort to provide application developers with an easy to use, application independent method to describe, store, share and manipulate graphical objects. GDL does this well. However there are limitations to the system. The components and focus of GDL are still relatively low-level and platform specific. Many of the GDL tokens correspond closely to Silicon Graphics GL functions [Ref. 12]. This still requires the developer to understand the use and interactions of many GL functions. A higher level of abstraction, including more high level components, would be helpful in many cases. The abilities of the system described in [Ref. 10] coupled with the advantages associated with the language format of GDL would be ideal.

Another limitation is the size of the GDL library. As mentioned before, the typical linker must be forced to link the modules of the derived token classes. Since it is not known prior to execution which tokens will be needed, all derived tokens and their support modules must be linked to an application. This results in almost the entire library being linked to an application even though many tokens may not be used. This is more of a linker problem than an GDL problem but is still a consideration for developers, as the size of the resulting executable and the placement of token modules may affect the page fault and cache performance of the application.

Future work on GDL will try to address these issues. The primary focus of future work will be to add extended functionality. An important aspect of many objects within NPS applications is physical characteristics like mass, center of gravity, etc. Extensions to GDL that embody object characteristics for supporting physically based modeling might be very helpful in providing higher level modeling capabilities. Other high level graphics support for advanced features of the graphics system, like atmospheric effects, motion blur, and anti-aliasing, are also likely candidates for support by GDL.

A. CONCLUSION

GDL provides high level application support. It incorporates an application independent language for describing graphical objects as well as a medium to high level graphics interface system. It is simple to use, easy to extend and maintain, and very flexible and capable. GDL gives application designers the leverage needed to rapidly prototype and develop applications. The ability to create and maintain collections of object models as well as individual components that can be used in many different systems is critical to virtual world development at NPS.

APPENDIX A: NPSGDL USER AND DEVELOPER'S GUIDE

Introduction

The NPS Graphics Description Language allows the description, management and manipulation of graphical objects in an application independent format. The system can be utilized on a variety of levels satisfying the needs of most users. At the highest level, the NPSGDL system facilitates the use of ASCII files containing textual descriptions of graphical objects. The user can use these files as program objects in a simple and abstract way. Low level usage of NPSGDL allows access to individual classes of objects that represent different graphics entities, concepts and algorithms. At this level, NPSGDL provides a simple, higher-level interface to the Silicon Graphics Inc. (SGI) Graphics Library.

Assumptions

This manual makes the following assumptions and considerations about NPSGDL users, namely: a user is a programmer/designer familiar with graphics programming on the SGI IRIS/4D series graphics workstations, the user is familiar with the SGI graphics library (GL), the user is expected to be familiar with the C++ language and the concepts of classes, inheritance, polymorphism and encapsulation as implemented in C++. NPSGDL is implemented using the object oriented approach to design and programming and for this reason is implemented in C++. There is currently no C language interface to NPSGDL. In addition to assumptions indicated above for users of the system, a developer is assumed to be a designer/programmer tasked with maintaining and/or extending the NPSGDL system. Furthermore, a developer should be thoroughly familiar with all the concepts related to the user.

User and Developer's Guide Organization

This guide consists of four major sections. Each major section has several subsections. Where applicable, the subsections subject matter and content will begin at a level for the general user and progress to more technical details for the developer.

The first section is a system overview. It describes the structure, capabilities and normal usage of the NPSGDL system. The second section describes what a user must do in order to use NPSGDL in his/her applications. It discusses linking to the libraries, file dependencies and file format compatibility. The third section discusses the NPSGDL Support Classes. NPSGDL uses several special classes to provide advanced capabilities. Many of these classes can be used by users in their applications without directly using NPSGDL. This section discusses those classes. The final section is the NPSGDL Class and Language Description. This section details each of the classes that make up the NPSGDL system. Each class's data members, methods and normal behavior is discussed with examples of usage from description files and directly within user program code.

Author and Point of Contact

The NPSGDL system was designed and implemented by Lt. Kalin Wilson in 1991-1992 as part of a Master's Thesis.

Suggestions for improvement, questions and bug reports are welcome. Please direct them to the permanent point of contact below:

Dr. Michael J. Zyda
Naval Postgraduate School
Department of Computer Science
Code CS/ZK
Monterey, Ca 93943
zyda@cs.nps.navy.mil

NPSGDL SYSTEM OVERVIEW

Purpose

The purpose of the NPS Graphics Description Language is to provide an application independent method of storing, retrieving and managing graphical objects and components. NPSGDL objects are normally described in ASCII files in a simple, easy to read syntax. Any ASCII text editor can be used to create or modify an NPSGDL object. The application independent nature of the NPSGDL system offers great flexibility to the application developer. Graphical objects displayed and manipulated by application code can be modified without the need for recompilation. A key advantage of NPSGDL is the ability to create collections of ready-made objects that any application can use. This facilitates rapid prototyping and sharing of objects across applications.

General Structure

NPSGDL is primarily a description language system. As such, the normal use of the system is through ASCII description files. NPSGDL is organized as language tokens. Each token in the language represents a separate graphical entity or algorithm. An example would be the **Defmaterial** token which describes lighting material characteristics or **Defpoly** which describes a filled polygon. Each token falls into one of fi , general token categories:

1. **Deftokens**: These are tokens that define system variables or settings that can be used by other tokens as needed.
2. **Settokens**: These are tokens that establish the current graphics context. They are normally paired with **Deftokens**. An example is **Setmaterial** which enables the use of a previous **Defmaterial** definition for rendering.
3. **Drawtokens**: These tokens represent visible entities that are rendered to produce the object shell displayed for the user.
4. **Xformtokens**: These tokens represent graphics transformations used to alter the display of an object at run time. An example is **Translate** which applies a

normal xyz translation to an object.

5. **Othertokens**: Tokens in this category do not affect the graphics process. They are provided for documentation and convenience. Examples are the **Comment** tokens and the **Name** token used to name an object.

Each token has it's own syntax or format that it recognizes in a description file. A token's format can consist of zero or more fields which represent token component values. Some formats are open ended relying on a number to specify the number of components in the format. Other token formats are closed and have a field that marks the end of the format. This is illustrated in Figure A.1 . **Token** formats are string and line oriented. Each token parses itself. Normally a token will parse by reading a complete string, up to whitespace, and then trying to recognize the string. The parse process is line oriented in that some tokens expect to find values or strings on the same line as other strings or fields. In Figure A.1 it would be a parse error if the string "shinyblue" appeared on the next line. The **Defmaterial** token expects to find it's name on the same line as it's identifier. In general the format of a line in a token format should be adhered to. The ordering of the lines of a closed format is not significant. In the **Defmaterial** definition in Figure A.1 , the lines for the ambient, diffuse or shinyness fields could be in any order.

```
// the defmaterial format is closed as are most deftokens
defmaterial shinyblue
ambient 0.3 0.45 0.95
diffuse 0.1 0.3 0.75
shinyness 30.0
defend

// the defpoly format is open. It relies on user information
defpoly high
0 0.7675 0.7675
3
0 0 0
0 10 0
0 0 10
```

Figure A.1 **Token** Formats Example

In most cases NPSGDL is case insensitive. The exception is names given to or referred by tokens. The **Defmaterial** name "shinyblue" shown in Figure A.1 would need to be matched exactly by any **Setmaterial** trying to use that definition.

Format errors are reported as NPSGDL warnings on cerr, the standard error output stream. Errors in formatting usually result in warnings that say that some token name is not recognized. Errors in name mismatches will result in a warning that reports that the given name was not found in some table. In general, the warning issued by NPSGDL will be informative enough to trace the source. Also when an error occurs, a reasonable default is used if possible to allow execution to continue otherwise your program is aborted.

As much as possible orthogonality and reasonable defaults are used to make the language easy to use. This means that tokens with similar formats use similar names and values. Also, not all fields need to be specified. If the default value is adequate you do not need to specify it in the description file. See the individual token descriptions for details.

Each token is represented by a C++ class. Most of the classes are relatively self-contained. This makes it possible to use NPSGDL has an alternate interface to the GL library adding some object-oriented properties and higher level capabilities. Application developers can use the token classes directly within application code either with or without object description files.

Capabilities

NPSGDL can be a very powerful tool. It offers a wide range of tokens with differing functions. These can be combined in almost any fashion to achieve different graphics effects. In most cases NPSGDL objects are used for simple display of a single graphical object such as a house or car. NPSGDL provides many tokens that encapsulate GL functionality. So many of the things that you normally do with the GL you can do in an NPSGDL description file.

NPSGDL has many advanced capabilities as well. These are discussed below and include: high level objects, high level algorithms, subobject management, multi-resolution display, simple animation facilities and simple viewpoint control.

High Level Objects

Having simple to use, ready made, high level objects makes application object development much easier. NPSGDL provides parametrized circles, spheres, cylinders, cones and cubes. Using these objects you can create many complex, visually pleasing objects in little time.

High Level Algorithms

There are many graphics algorithms that are difficult to remember or implement. NPSGDL currently supports one of these, polygon decaling [Ref. 21]. Polygon decaling is a process to display two or more coplanar polygons using z-buffered hidden surface removal. Normally if you have coplanar polygons in the presence of z-buffering the polygons will appear to shimmer or tear on top of each other as the z-buffer tries to resolve which is on top of which. Decaling is an algorithm to overcome this problem. NPSGDL provides the Defdecal token which allows you to specify two lists of tokens, the underlay list and the overlay list. When displayed, the Defdecal takes the steps neccesary to decal the overlay list onto the underlay list. This is very helpful for things like roads, opaque windows and the like.

Subobject Management

Few interesting graphical objects are simple enough to not have or need subobjects or components. NPSGDL provides the Defobject token to help manage and facilitate the use of subobjects. A Defobject is named and contains a list of any tokens, normally these are displayable. This Defobject is then stored in a table accessible by Defobject name. Any other NPSGDL object can call or display a previously defined Defobject by name. The subobject can be transformed in the calling object as needed without affecting the subobject description. This method of subobject management makes it easy to build collections of commonly used object components and then combine them as necessary. A good example of this is a jeep or other vehicle. Each of the tires is identical except for location. We can store the definition of the tire in a Defobject. The jeep object would then call the tire subobject four times translating the tire to the proper location for display each time.

NPSGDL does not provide any automatic facilities to manage object-subobject relationships. Any relationships, graphically or behaviorally are handled by the user either in the description file or in the application code.

Multi-Resolution Display

Many complex graphics applications are graphics bound in performance. In these cases a goal of the developer is reducing the number of polygons displayed during each frame while maintaining visual quality. NPSGDL helps support this goal by providing two multi-resolution display schemes for graphical objects. This is typically done by providing a means to selectively display different parts or versions of an object during run-time. An example would be to display low detail views of an object when it is far from a viewer and increase the level of detail as it gets nearer.

Each NPSGDL Drawtoken is given one of three resolutions: High, Medium and Low, as assigned by the user. The default resolution is High. Within a NPSGDL object, Drawtokens are sorted by resolution for efficient display. How the resolutions are used to display an object depends on the mode of the display. Currently, NPSGDL supports two modes, hierarchical or single. The default display mode is single.

The single resolution display mode displays only the tokens with a resolution matching the resolution provided to the display method of an object. This mode is the simplest to use and provides for fast efficient display. Also this mode provides for the specification of three distinct display models within one object description file. The prime disadvantage is the possible duplication of token definitions between the different resolution models within an object.

The hierarchical display mode displays all tokens with a resolution equal or lower than the resolution provided to the display method of an object. For example, displaying an object in high resolution displays all tokens while displaying an object in medium resolution displays only medium and low resolution tokens. This display mode allows the user to specify which parts of an object will be visible at the different resolutions. It also helps reduce the number of polygon duplications within an object. The prime disadvantages to this mode are complexity and efficiency. Designing an object that is displayed hierarchically and looks good at the various resolutions is not easy. Just providing a shell and then specifying detail features as high resolution tokens usually results in small, noticeable holes in the shell. However, if the time is taken to design a hierarchical model, it will normally be smaller than a corresponding single resolution model (with three versions). This translates to disk space and run-time memory savings. The efficiency issues with hierarchical display are small but possibly important. As was pointed out above, the Drawtokens are sorted by resolution. To preserve the behavior and appearance of the object after the sort, Settokens and Xformtokens are duplicated within the display lists. This could result in duplicate calls for material bindings etc. during display. For example given the following object description fragment in Figure A.2 , the system binding to activate the blue Setmaterial would occur three times, once for each resolution. Normally this is trivial but is important to understand.

```
Setmaterial blue
// assume material definitions exist
defpoly high
1 0 0
3
0 0 0
0 1 0
0 0 1

defpoly medium
0 1 0
3
1 1 1
1 2 1
1 2 2

defpoly low
0 1 0
3
2 3 3
1 1 1
0 0 0
```

Figure A.2 Multi-resolution Example

Animation

One of the most powerful facilities provided by NPSGDL is management and control of simple animation for NPSGDL objects. The NPSGDL animation system is simple to use and very flexible. It allows the user to define named animation variables that can be used as values in various NPSGDL tokens. These animation variables, called **Defvariables** in NPSGDL, can be updated either completely by NPSGDL or by the user and NPSGDL. Once a **Defvariable** is defined it can be used by many tokens, notably the **Xformtokens** and some **Drawtokens**. Using animation variables, the user can turn the control of animating such things as turning wheels, rotating antennae, spinning propellers and a myriad other objects that only require simple continuous animation over to the NPSGDL system. For discontinuous or more complex animation, NPSGDL provides hooks to allow the user control over updating the animation variables. Previously the user would have had to manage the separate components of an object that needed to be animated within application code.

Viewpoint Control

NPSGDL provides facilities for specifying and using viewing volumes. Two categories of tokens work together to provide this: cameras and viewpoints.

Two types of cameras are provided by NPSGDL, orthographic projection and perspective projection. Camera tokens specify the dimensions of a view volume. Cameras are named and stored in a table for random access. This simplifies the management and use of multiple view volumes. Cameras can be turned on and off by name.

Viewpoints specify the location and orientation of a view volume. Like cameras, they are named and stored in a table for random use. Viewpoints are very flexible. The from and to values can be specified either absolutely, using animated variables, or by object name (the object origin is used).

Many users can not need the viewpoint control facilities. Many simple graphics applications use one viewing *volume that is fixed.* By specifying the view volume with a camera and viewpoint in a description file, the user gains the ability to change the volume dimensions and orientation without recompilation. The combination of viewpoint control and simple animation makes it possible to define simple animation sequences completely within an NPSGDL description file. While this can not be normally applicable, the system is very flexible and many possibilities exist.

All of the facilities above will be discussed in more detail with examples in the token descriptions that follow.

USING THE NPSGDL SYSTEM

NPSGDL gives the application developer many powerful and helpful tools to speed prototyping and development. This section discusses the essentials of actual use of NPSGDL. Topics include:

Compiling, linking and debugging NPSGDL applications.

Essentials of object use.

Considerations for NPSGDL object use.

NPS Object File Format (OFF) compatibility.

Compiling, Linking and Debugging NPSGDL Applications

Using the NPSGDL system is quite straightforward and easy. There are a large number of object description files, in both GDL and OFF format, available for general use. Using these ready made objects, users do not need to learn the details of the language system itself. This section will present the steps needed to produce a working NPSGDL application. The examples shown assume that the object description files already exist and are not shown.

NPSGDL is written in C++ using C++'s object oriented capabilities. No C language functions are provided at this time. Therefore NPSGDL applications must be compiled and linked using the C++ language system. As of this writing, NPSGDL was compiled under Irix 4.0.x using the Silicon Graphics Inc. port of AT&T cfront 2.1.

The primary concerns for easy use of the NPSGDL components are the locations of the header and library files. These locations should be specified to the compiler via the -I and -L switches. The main paths for NPSGDL files are:

/n/gravy1/work/zyda/NPSGDL for the header and source files.

and

/n/gravy1/work/zyda/NPSGDL/lib for the object module libraries.

NPSGDL relies heavily on header files and modules in the NPSCL C++ library. NPSCL is a general purpose collection of standard data structure classes, templates and systems. It is available for general use and is documented separately. See the point of contact listed above for details. The locations for the NPSCL headers and libraries are:

/n/gravy1/work/zyda/NPSCL for the header and source files.

and

/n/gravy1/work/zyda/NPSCL/lib for the libraries

There are three library versions available for NPSGDL. One, libNPSGDL.a, has been optimized using the -O2 compiler switch. The other, libDNPSGDL.a, was compiled with the -g compiler switch to include debugging information in the object code. While debugging your NPSGDL application, it can be useful to link to the debugging library to trace any errors that can occur within NPSGDL. A third library, libDTNPSCL.a, available for developers, contains support for a debugging trace log using the NPSCL debugging classes in "`debug.H`". Programs linking to this library must initialize the debugging system with the FTRACER macro as described in the NPSCL manual. Failure to do so will cause a program abort and core dump.

NPSGDL uses several other libraries. The NPSCL C++ class library (see separate documentation) provides many of the basic data structures used by NPSGDL. NPSCL also provides the run-time typing and reference counting support systems discussed later. The NPS image support library is used to provide image manipulation capabilities for texture support. The image support library in turn requires the SGI image library. The standard gl shared library is used as is the standard math library. These libraries are listed roughly in the order they should be included when linking. The normal UNIX linker does a single pass through libraries to resolve references. Thus the NPSCL library should be listed after the NPSGDL library because the NPSGDL library will generate references to the NPSCL library objects.

The easiest way to manage a NPSGDL project is using the UNIX *make* utility. If you are not familiar with *make*, refer to the online manual pages or any good UNIX book. A simple *make* file is given below that illustrates how to use the compiler directives to locate header and library files. Simple comments are included in the *make* file. See the manual pages on *make* for detailed information. One thing worth mentioning is the *makedepend* utility. Usually one of the hardest and most often neglected things to do when using make files is properly specifying inter-file dependencies. The makedepend utility automates this task by processing project source files as the compiler will and building dependency lists that it appends to your *make* file. Then using *make* implicit rules, you can almost fully automate the compilation and linking process. For more information on *makedepend* consult the manual pages.

```
# This is a sample makefile used to compile
# and link an example NPSGDL application

# These are macro definitions that can be used
# by other statements in the file

# define the standard places to look for include (header) files
INCLUDE= -I/usr/include/CC -I/n/gravyl/work/zyda/NPSCL/lib \
-I/n/gravyl/work/zyda/NPSGDL

#define the standard places to look for libraries
LIBDIRS = -L/n/gravyl/work/zyda/NPSCL/lib \
-L/n/gravyl/work/zyda/NPSGDL/lib \
-L/n/gravyl/work/zyda/imagesupport

# This is used for show dependency on the library
DEPNPSGDL = /n/gravyl/work/zyda/NPSGDL/lib/libNPSGDL.a

# This application will be optimized
# -DNDEBUG is used by NPSGDL to turn off assertion checking
# you can or can not need it for your application
# -D__SGI__ is convenient if you need conditional code for SGI

DBGFLAGS= -O2 -DNDEBUG -D__SGI__

# Some more descriptive names for the libraries
NPSCL_LIB= -lNPSCL
GL_LIB = -lgl_s -lm
NPSGDL_LIB= -lNPSGDL
IMAGE_LIB = -lnpsimage -limage

SRC = testoff.C

# tell make about the '.C' suffix
.SUFFIXES: .C .o

# This is an implicit rule that defines how to compile a .C file
.C.o:; CC -c $(DBGFLAGS) $(INCLUDE) $*.C
ALL = testoff
all: $(ALL)

touchall:
 -touch *.H
 touch *.C

clean:
 -rm -f *.o

# generate the dependency listing
depend:
 makedepend -f makefile -- $(DBGFLAGS) $(INCLUDE) -- $(SRC)

# The normal dependency and compile command
testoff: testoff.o $(DEPNPSGDL)
 CC -o testoff testoff.o $(DBGFLAGS) $(INCLUDE)\
$(LIBDIRS) $(NPSGDL_LIB) $(GL_LIB) $(IMAGE_LIB) $(NPSCL_LIB)

# The dependencies will be listed below by makedepend
# DO NOT DELETE THIS LINE -- make depend depends on it.
```

Figure A.3 Sample NPSGDL application makefile

Essentials of object use.

The NPSGDL system is dynamically initialized at run-time prior to the `main()` function being called. This means that in most cases there is no user initialization of the system needed before using NPSGDL objects. Another important point is that NPSGDL is very robust. It uses reasonable defaults for both language token values and for object method parameters. The extensive error checking and recovery built into NPSGDL generally allows your application to work with faulty NPSGDL object files, easing the development process.

NPSGDL objects are represented by the NPSobject class. NPSobjects are essentially a list of Tokens that are initialized at run-time either from an object description file or manually by the user. The normal usage is to initialize an NPSobject from a file. This is the approach shown here. Manually defining an NPSobject is discussed in the class details section below.

There are a minimum of three steps that must be taken to display an NPSobject: construction/initialization, definition and display. NPSobjects are constructed in three different ways. An empty NPSobject can be constructed using the default constructor and the syntax shown in Figure A.4 .

```
#include "NPSobject.H"
#include <gl.h>

main() {

winopen("example2");
RGBmode();
gconfig();

// create an empty object
NPSobject firstobj;

// create an object and initialize from a file
NPSobject secondobj("secondobj.gdl");

// initialize the first object
firstobj.read_from("firstobj.gdl");

// define the objects
firstobj.define();
secondobj.define();

// put up a view volume and any transfomations
perspective(450,1.25,10, 1024);
lookat(0, 0, -150, 0, 0, 0, 0);

// display the objects
firstobj.display();
secondobj.display();

// exit
}
```

Figure A.4 Simple use of NPSobjects

The most convenient way to construct an NPSobject is to provide a valid NPSGDL object description file name to the NPSobject constructor as shown in Figure A.4 . When this constructor is used four things happen. First, the named file is opened for reading. Second, the tokens in the file are read to initialize the list of tokens in the NPSobject. Third, the NPSobject is optimized. Lastly, the description file is closed. Optimizing an NPSobject involves sorting the Drawtokens into resolution lists and Deftokens into a definition list. These special lists reduce the number of tokens traversed for various operations giving a small performance boost. Using the filename constructor or the `NPSobject::read_from(char *filename)` method ensures that NPSobjects get optimized. If you use any other means of NPSobject construction/initialization you should ensure that the object is optimized.

After an NPSobject is initialized it must be defined. The `NPSobject::define()` method does this. The definition process traverses the tokens that make up an NPSobject and defines any Deftokens found. As mentioned before, Deftokens represent system settings that are stored for random access by other tokens. Many Deftokens contain information needed by the SGI GL. The `define()` method ensures that all system information is initialized before the rendering process.

Once all NPSobjects have been defined, they are ready to be displayed. Providing that no errors occurred during construction, initialization, or definition and that all Settokens use valid names, your NPSobjects should display themselves when you invoke `NPSobject::display()`. The display method for NPSobjects has a single parameter that specifies the display resolution. The default for this parameter is HIGH. You can specify HIGH, MEDIUM, or LOW to display an NPSobject at various resolutions. As mentioned above the default display mode is single resolution.

That is basically all there is to using NPSGDL. Figure A.4 shows the basics of what is needed and how it looks. The essential steps beyond those normally required for a graphics application are:

1. Include "`NPSobject`.H"
2. Construct an NPSobject giving a filename.
3. Ensure a valid GL window is open and active.
4. Define the NPSobject.
5. Set up the graphics loop.
6. Display the NPSobject when needed at various resolutions.

Considerations for NPSGDL object use

There are several things that you should be aware of when using NPSobjects. The first is using global objects. NPSGDL uses a highly dynamic scheme to initialize the language system at run-time without user intervention. It does this using a combination of special constructors in each of the token classes and static objects. The static objects enter token information into a system table that is used to read description files. While this is powerful, there are drawbacks from the user's point of view.

The first drawback stems from the use of static objects. The C++ language gives minimal guarantees concerning static object construction. The only guarantee given is that all global objects will be constructed prior to the invocation of `main()` or their first use in the module they are declared in. For this reason you can not construct global NPSobjects and initialize them from a file. This is because the system table that contains token information can not be created or initialized before your object is constructed. Thus, the only way to facilitate global NPSobjects is to construct empty objects or use NPSobject pointers and initialize the objects within the scope of `main()`.

The second drawback of the dynamic initialization scheme of NPSGDL is discussed in detail in the body of the accompanying thesis. In essence, normal use of an NPSobject does not generate references to NPSGDL token modules so the linker does not link them in. To remedy this, a special class is used to register each token class, generating the unresolved references needed to link the token modules to your application. The consequence of this is that most or all of the NPSGDL token modules get linked to your application whether you use them or not. A method was devised to allow users to selectively link to the entire NPSGDL library or individual modules based on the tokens used in application object description files. However, actual use of this method was cumbersome and often resulted in confusion. For this reason, the automatic method is used. In general, you should not see any effects on your program from this method except for executable size.

Another consideration for using NPSGDL objects is that they require that a valid, active window be open to receive SGI GL requests and commands. Many of the NPSGDL token define and display methods call GL functions that issue commands affecting the graphics system. These functions are valid only if a window is open.

Note: *If you don't open a window before defining or displaying an NPSobject your program will most likely cause a graphics error and core dump.*

NPS Object File Format (OFF) compatibility

NPSGDL is an outgrowth of the NPS OFF system. As such, NPSGDL will read and use OFF object files that are in the correct format. The correct format is specified in Book Seven of Dr. Zyda's graphics notes series. NPSGDL extends the format of many of the NPS OFF tokens. When reading an OFF file, NPSGDL will issue a warning when it expects a value that the OFF format does not provide. A reasonable default value is used in these cases. To ease the use of OFF files with NPSGDL, a simple conversion utility is provided in the NPSGDL directory. The utility, `off2gdl`, will read an OFF file in any valid format and convert it to a GDL file using the default values for any missing parameters. You can then edit the GDL file with any ASCII text editor to modify the object descriptions. The syntax for using `off2gdl` is shown below in Figure A.5 .

Note: *In general you should use different names for the source and destination files to preserve the integrity of your source file.*

```
off2gdl source_off_file.off dest_gdl_file.gdl
(extenders optional)
```

Figure A.5 Using the off2gdl file conversion utility

NPSGDL SUPPORT CLASSES

NPSGDL classes utilize many relatively simple classes to implement low-level behavior. These support classes are mostly either concrete data types (classes that act like built in types) or instantiations of template classes. A separate collection of classes named NPSCL (NPS Class Library) provides most of the support classes that NPSGDL uses. Other classes were designed specifically for NPSGDL but can be generally useful. This section serves two purposes. One is to introduce the support classes so that you are familiar with their names and functions. Subsequent references to these classes in later sections assume that you have read this section. Secondly, the classes used to support the NPSGDL token classes that are not part of the NPSCL library are discussed in detail. These classes provide capabilities that will likely be useful in both NPSGDL and non-NPSGDL applications. Also developers maintaining or extending the NPSGDL system will need sound knowledge of these classes and their capabilities.

The NPSCL Classes and Templates

The NPS C++ Class Library was created as a basis for NPSGDL development. As such, NPSGDL uses it's component classes extensively. The classes in NPSCL are either stand alone concrete data types or class templates. A concrete data type is a user-defined data type that has methods and operators defined that allow it to behave well on it's own, much like the built-in types. Class templates are generic class definitions that can be instantiated with different types or classes. Templates are very useful to define container classes, classes that hold other classes like a linked-list. NPSCL implements it's template classes using regular C macro definitions relying on the preprocessor for macro expansion. The issues involved with this are discussed in the NPSCL manual. When discussing use of the template classes, I will use the macro template syntax instead of the standard C++ template syntax. When NPSGDL was implemented, a C++ compiler with template support was not available. Future modifications to NPSGDL and NPSCL will likely add true template support. For our purposes, the only difference in syntax between C++ templates and macro templates is that C++ templates use the angle brackets, '< >', and macro templates use parentheses, '()'. The macro template syntax can be confusing at first because the template name and type parameter look like a function call. Occasionally it is necessary for a user or developer to use the actual type name of a template instantiation rather than the template syntax. This is necessary because the preprocessor does not expand the nested macros. Figure A.6 illustrates this.

```
#include "genarray.H" // NPSCL array template
#include "genlist.H" // NPSCL linked list template
#include "String.H" // NPSCL string class

// declare and instantiate a list of strings
DeclareList(string); // macro expands to class definition
ImplementList(string); // macro expands to class implementation
ImplementList2(string);

// Now we have lists of strings so the following is valid
List(string) my_slist; // create a list of strings

// List(string) actually expands to List_string for the compiler

// now lets make an array of lists of strings
DeclareArray(List_string); // must use actual class name
// DeclareArray(List(string)) doesn't expand properly
ImplementArray(List_string);
ImplementArray2(List_string);
ImplementArray3(List_string);

Array(List_string) listarray; // create an array of lists
Array(List(string)) listarray; // macro expansion error!

// put a string in the list and add the list to the array
my_slist.add_at_tail("Aren't Templates Fun?");
listarray.push(my_slist);
```

Figure A.6 Template Usage Fragment

The classes used by NPSGDL include string, Date, Array(T), List(T) and HashTable(K,T). You should consult the NPSCL manual for details on the methods available and behavior of these classes.

Note: *In later sections the above names will be used to refer to NPSCL classes. Standard C strings will be referred to as 'C strings' or char*. The 'T' parameter for the template classes represents any valid type.*

NPSGDL Low Level Classes

There are template instantiations and stand alone classes that NPSGDL uses that can be generally useful. The template instantiations involve Array(T)'s and reference counted pointers, Refptr(T). Reference counted pointers are discussed in detail in a later section as well as in the body of this thesis. These pointer classes are used in the Deftoken classes to manage Deftoken instances in the corresponding table. All of this is discussed below. When working with Deftokens directly from application code, use reference pointers to maintain the integrity of the simple garbage collection system used by NPSGDL. By including the header file for a Deftoken, you gain access to the Refptr for that class. So if you need to use a Refptr for a Deftoken, remember that they are already available through NPSGDL. Refptrs for other NPSGDL classes, except NPSobject and Token, will need to be instantiated explicitly. See the NPSCL Users Manual for details on using the Refptr(T) template.

NPSGDL instantiates the Array(T) template for the types float, vertex and vertex2. Vertex and vertex2 are discussed below. As described in the NPSCL manual, the Array(T) class implements a dynamically sized/resized, robust array of objects with many sophisticated capabilities. There are many applications in graphics for an array of floating point numbers. Likewise for arrays of vertices. If you find yourself needing these objects, remember that NPSGDL already provides them in it's library. The header files for these objects are shown below in Table A.1 .

Table A.1 Predefined Array Classes

Array Class	Header files
Array(float)	farray.H
Array(vertex)	varray.H
Array(vertex2)	v2array.H

Many graphics applications involve defining and manipulating object vertex coordinates. All locations within a coordinate system can be thought of as a single vertex, either as part of an object like a polygon or as a point location like the origin. Likewise every vertex can be represented by a vector from the origin of the coordinate frame to the vertex location. NPSGDL uses vertex and vertex2 classes to represent and manipulate vertex coordinates. The vertex class does double duty by providing simple vector operations like length, and dot and cross products. The vertex class represents three dimensional vertices while the vertex2 class is two dimensional. The vertex classes provide methods to access the coordinate values by name or array subscript style. Also vertices can be added, subtracted, scaled as well as read from and written to streams. They are versatile and useful in many contexts. Figure A.7 and Figure A.8 show the public interface of the vertex class. The vertex2 interface is identical (parameters will be vertex2 types, etc.) with the exception of the vector operation methods. Figure A.9 shows a code fragment illustrating the use of the vertex class

```
/* this file contains the definition of class vertex. A vertex
contains the coordinates
for a 3D point.
*/

class vertex {
public:
  inline vertex();
// construct an empty vertex = 0,0,0

vertex(float x, float y, float z);
// construct a vertex from given values

vertex(const vertex&);
// construct a vertex from another

vertex& operator=(const vertex&);
// assign the value of a vertex to this

inline float& operator[](int i);
// access xyz values by array index (range checked)

int operator==(const vertex& v) const;
// compare vertices for equality (all members equal)

int operator<(const vertex& v) const;
// compare vertices for order
// (all members of this less than v)

vertex operator+(const vertex& v) const;
// add the contents of v to this and create a new vertex

vertex operator+(float add);
// non destructive addition of a value to each coordinate value

vertex operator-(const vertex& v) const;
// subtract the contents of v from this and create a new vertex

vertex operator-(float add);
// non destructive subtract of value from each coordinate
vertex operator*(float sc) const;
// non-destructively scale the value of a vertex

vertex operator/(float div) const;
// non-destructively scale the value of a vertex

vertex& operator +=(float inc);
// destructive addition of vertices

vertex& operator -=(float inc);
// destructive subtraction of vertices

operator float*() { return v;}
// provides normal C array behavior. Use with caution

float distance(const vertex&) const;
// return the distance betwen this and the arg
```

Figure A.7 Class Vertex Declaration

```
vertex midpoint(const vertex&) const;
// return the midpoint on the line defined
// by this and the argument

vertex cross(const vertex&) const;
// Return the vector cross product of this and the arg

float dot(const vertex&) const;
// returns the vector dot product of this and the arg

float length() const;
// returns the vector length of this vertex

float x() const { return v[0];}
// read the x coordinate by name

float y() const { return v[1];}
// read the y coordinate by name

float z() const { return v[2];}
// read the z coordinate by name

void x(float x) { v[0] = x;}
// write the x coordinate by name

void y(float y) { v[1] = y;}
// write the y coordinate by name

void z(float z) { v[2] = z;}
// write the z coordinate by name

  void read_from(istream&);
// read the coordinate values from a stream

void store_on(ostream&) const;
// write the coordinate values on a stream

};

ostream& operator<<(ostream&, const vertex&);
istream& operator>>(istream&, vertex&);
ostream& operator<<(ostream&, const vertex*);
istream& operator>>(istream&, vertex*);
```

Figure A.8 Vertex Class Declaration (cont.)

```
#include "vertex.H"

// create two vertices. one at 0,0,0 one at 10,10,10
vertex v1, v1(10, 10, 10);

// read the first vertex from an open istream
cin >> v1;
// translate v1 by v2 saving v1
vertex v3 = v1 + v2;
// compute the normal to the plane defined by v1 and v2
vertex n = v2.cross(v1);  // v2 into v1

// write the vertices to an ostream
cout << "v1 = " << v1 << endl;
cout << "v2 = " << v2 << endl;
cout << "v3 = v1 + v2: " << v3 << endl;
cout << "normal of v2 into v1 is " << n << endl;

// if the y value of the normal is negative make it positive
if(n.y() < 0){
n.y(-n.y());
// better yet recompute the x product
n = v1.cross(v2);
}
```

Figure A.9 Sample of vertex use

NPSGDL SYSTEM DETAIL

This section discusses the low level details of the NPSGDL system. The details of how objects are used, class relationships, object and module interactions and the special subsystems used by NPSGDL are presented. Specific descriptions of class data and method details are discussed in the next section, NPSGDL Class Details. This section is more technical than prior sections and assumes a solid basis in C++ and object oriented concepts. The information in this section is not necessary for common users, it is tuned more towards developers who are maintaining or extending NPSGDL. However if you have the background and interest, the information in this section will help you understand how NPSGDL works, helping you debug your NPSCL applications and object description files.

The C++ language supports many paradigms of programming. It is object oriented constructs can be used on many levels. NPSGDL is highly object oriented and uses all of C++'s object oriented constructs extensively.

The NPSGDL library is made up of a Token hierarchy and the NPSobject class. The NPSobject acts as a container/manipulator of Token objects. The Token hierarchy provides the functionality of the system. Several subsystems are employed by NPSGDL to provide object persistence, reference counted garbage collection, and run-time type checking. Each of these components is discussed in detail below.

The NPSobject

The NPSobject has been discussed in previous sections of this guide. So you can have already seen some of this material. The NPSobject is essentially a collection of Tokens. This collection is organized as a doubly-linked list of Token Refptrs. Refptrs are reference counted pointer objects. These are objects that behave like pointers with the added ability to track pointer references to objects and automatically destroy the object when it's reference count goes to zero. This is discussed in detail in the section concerning the garbage collection subsystem. For now, just remember that a Refptr is like a normal pointer used in the garbage collection system. Notice that the list of tokens in the NPSobject is a list of pointers to Token objects. The Token class is the base class for the Token hierarchy. The NPSobject manages all Tokens generically, that is it is normally not concerned with the actual type of Token attached to a Token Refptr. This method of object management is the core of most robust object oriented system. Since the NPSobject only needs to know about Tokens, we can add new Tokens to the hierarchy and have them work perfectly with the NPSobject. There are a few cases where the NPSobject needs to know the type of individual Tokens but these are few and in general you can think of the NPSobject as only having knowledge of generic Tokens.

Each NPSobject also has four special lists that are used to increase the performance of the class. When an NPSobject is optimized, it's main token list is traversed and the Tokens are sorted onto these special lists. The special lists contain pointers to Deftokens, high resolution Drawtokens, medium resolution Drawtokens, and low resolution Drawtokens. The Drawtoken lists will also have pointers to Settokens and Xformtokens to that the appearance of an object is preserved across the separate lists. These lists are generally shorter that the main list. Also the lists separate tokens by functionality. As explained below, not all tokens implement the entire Token interface. For example calling the display method on a Deftoken is a no-op but still costs a function call. The special lists help ensure that these no-op calls don't happen as they would if a single list was used. The single main list is still necessary to maintain the proper composition of the object in case the object is written to a file.

An NPSobject has additional data members that provide information to the user at run-time. This data is extracted from tokens contained in an object description file. If the file does not provide the information, default values are used but are not saved if the NPSobject is written. The information items are name and origin. The name of an object is specified by the Name token. If an object description file does not have a Name token, the name of the NPSobject will be empty. The origin of an object is specified by an Origin token. The default origin of an NPSobject is 0,0,0. Methods are provided to access the name and origin of an NPSobject.

The interface of the NPSobject class is similar to most other objects in NPSGDL. All objects in the system can read themselves from a stream, write themselves to a stream, define themselves, and display themselves. This makes using and extending NPSGDL simple and consistent. The define, display and store (write) methods on an NPSobject are implemented as simple traversals of one or more of the object's token lists forwarding the request to each token in the list. Polymorphism, through virtual functions, ensures that the correct behavior occurs based on the actual type of the token object attached to the base pointer.

Reading an NPSobject from a file is a bit more involved but not too difficult. As explained in the next part of this section, the Token class maintains a table of token information that allows tokens to be recognized on a stream. The Token class provides a method for sequentially retrieving tokens from a stream. This method returns a Token Refptr. When reading an object description file, the NPSobject simply opens an input file stream and calls the Token retrieval method until the end-of-file is encountered, storing the Token Refptrs in it's main list.

Occasionally it's important to know the boundary extent of an object. The get_bounds method provides this information. The get_bounds method traverses the main list looking for Drawtokens. Each Drawtoken is asked for a list of it's vertices. The vertex values are compared to current bounds values for the NPSobject to produce the NPSobject's rectangular bounds. The bounds information is stored in a `float[3][2]` array provided by the user. The bounds information is stored as xmin, xmax, ymin, ymax, zmin, and zmax.

The NPSobject provides most of the functionality that many users will ever need for application support. However, developers of tools and utilities that manipulate objects need more control over NPSobjects. For these users, the NPSToolobject is provided. The NPSToolobject is a subclass of NPSobject and thus retains the functionality of an NPSobject. An NPSToolobject adds method for adding and deleting tokens from an object, extracting groups of tokens, iteratively traversing the token lists, appending objects, and managing subobjects. These methods are relatively straightforward and are detailed in the class description later.

The Token Hierarchy

The Token hierarchy provides ninety percent of the functionality of the NPSGDL system. As discussed above and in the main thesis, the language recognized by the system is broken down into tokens. Each token can be composed of zero or more fields. Each token represents a different graphical entity like a material definition or polygon. Each token in the language is represented by a separate class in the Token hierarchy. The Token class interface specifies the common behavior for all tokens. Virtual functions are used to allow dynamic dispatch of function calls (messages) so Token pointers can be used to manage derived tokens. The Token class doesn't provide very much functionality. It provides default methods so that derived classes aren't required to override the entire Token interface. Most of these methods are empty (no-op).

Each Token has a unique integer identification that is initialized during construction. This id is used to test for Token identity verses equality, which is normally based on the contents of a token. A method is provided to retrieve the id of a Token. This can be used to maintain uniqueness when required.

The Token class manages the NPSGDL persistence subsystem. This is discussed in detail later. The basis of this system is a table, the tokentable, contained in the Token class that relates token names to addresses of methods that construct a token from a stream. This table is initialized dynamically at run-time. Each token in the hierarchy registers it's information with the tokentable using a special constructor and a static instance of the token. This scheme makes it easy to extend the NPSGDL language by adding new tokens. The disadvantages are related to linker technology and have already been discussed. Along with the tokentable, the Token class provides a method which sequentially reads tokens from a stream using the information in the token table.

Every token in the Token hierarchy shares a common interface. Most of methods in this interface are overridden by derived classes to provide the correct behavior. Where overriding is not needed, the Token class provides the default behavior. The interface and expected behavior are shown inTable A.2 . Using this interface in conjunction with virtual methods, clients of the Token hierarchy, like NPSobject, can deal with tokens generically. The power of this is evident in the simplicity of the NPSobject implementation and the ease with which new tokens can be added to the system. Adding new tokens to NPSGDL is shown in detail at the end of this section.

Table A.2 The Common Token Interface

Method Name	Description
Default constructor	Initializes an empty token.
Copy constructor	Initializes a token from another.
Reader constructor	Initializes a token from a stream.
Table init constructor	Registers information in tokentable.
void read_from()	Reads token description from stream.
void store_on()	Writes token description to stream.
void define()	Defines a token to the system if necc.
void display()	Displays a token if applicable.
void operator=()	Copies the contents of another token
	compatible type.
bool operator==()	Compares two tokens for equality.
bool operator<()	Compares two tokens for ordering
Token *clone()	Creates a distinct copy of a Token.

The Token class has five direct subclasses that serve to categorize derived tokens. These classes are Deftoken, Settoken, Drawtoken, Xformtoken, and Othertoken. These category classes serve two purposes: provide functionality common to all of their base classes and provide a common umbrella for type classification. The run-time type information system has the ability to test a type for membership in a hierarchy. For example you can ask the type system if a Defpoly token is a Drawtoken or a Token. Both of these queries will be true. So even though the classes in a category can not share any common functionality, as with Othertokens, the category class serves to group related classes.

Deftokens

The Deftoken class represents all of the tokens that represent system definitions. System definitions are things that are stored for random access by the user or other tokens. Some Deftokens represent entities that the SGI graphics system must know about before they can be used. These tokens represent things like materials, textures, and lights. Most Deftokens use a closed format that ends with a `defend` field. The exception to this is the Readobject token. The Deftoken class is responsible for reading and writing the name given to the token. It expects the name to be given on the same line as its identifier.

Each Deftoken subclass contains a static HashTable(K,T) which contains Refptrs to token instances. These tables are keyed on the token's name. These tables are normally used by other tokens to access the definition information of a Deftoken. However, they are declared public so any user can access the Deftoken tables. This can be useful to make a system wide change to a definition or some other manipulation of a class of Deftokens. The interface for the HashTable(K,T) template class is presented in the NPSCL manual.

All Deftokens provide a `define()` method which overrides the default method. None of the Deftokens override the default `display()` method. Table A.3 lists the current Deftoken subclasses.

Each Deftoken puts a default instance into it's table named "default". This instance is used by the system when an error occurs in a corresponding Settoken.

Table A.3 Deftoken Subclasses

Defmaterial	Deflight	Deflmodel	Defcolor	Defobject
Deftexture	Deftexenv	Deftexgenalg	Defvariable	Defocamera
Defpcamera	Defviewpoint	Readobject	Defsound	

Settokens

The Settoken class is closely related to the Deftoken class. Settokens are normally paired with Deftokens and provide the means to use previously defined information. Many Settokens relate to GL functions that affect the state of the rendering pipeline like activating a material or color. The Settoken class is responsible for reading and writing the name given to the token. This name must match the name of a previously defined corresponding Deftoken. If the Settoken name does not match any corresponding Deftoken name, the name "default" is used. This name will match a valid definition.

Most Settoken formats consist of a single line. Following the Settoken identifier is the name of the Settoken and optional state information. Many Settokens can be turned on and off. This feature allows you fine control over object behavior.

All Settokens define the `display()` method. The `display()` method is used to activate the behavior of a Settoken. The `define()` method for Settokens is inherited from the Token class providing the default behavior, which is a no-op. Table A.4 lists the current Settoken subclasses.

Table A.4 Settoken Subclasses

Setmaterial	Setbackmaterial	Setlight	Setlmodel
Settexgenalg	Settexenv	Setocamera	Setpcamera
Setcolor	Settexture	Setviewpoint	Setsound

Drawtokens

The Drawtoken class categorizes the tokens that have visible displays. These tokens represent the graphical entities that make up the visible part of an object. All Drawtokens have a resolution that is used for multi-resolution display management. It is the responsibility of the Drawtoken class to read and write the resolution of a Drawtoken. The default resolution is High.

Drawtoken formats are both open and closed. The tokens that represent lower level primitives are normally open like the Defpoly example in Figure A.2 on page 58. Other Drawtokens, like Defsphere, have closed formats.

All Drawtokens implement the `display()` method. Like Settokens, the `define()` method is inherited and is a no-op. Also, as noted above, a GL window must be active to receive the display commands from the Drawtoken. Table A.5 lists the current Drawtokens.

Table A.5 Drawtoken Subclasses

Defpoly	A_Defpoly	Defsurface	Deftmesh	Defqstrip
Callobject	Defdecal	Defline	Defsphere	Defcylinder
Defcone	Defcircle			

Xformtokens

The Xformtoken class serves as base to those tokens that alter the transformation matrices used to render objects. Xformtokens are not named and don't share any common behavior. Therefore the Xformtoken class has no responsibility to it's subclasses. Several of the Xformtokens are animated tokens. This means they accept animated variables, Defvariables, as one or more of their parameters. The use of animated tokens is powerful as was discussed above.

Xformtokens define the `display()` method to implement their behavior, using the default `define()` method.

Xformtokens are closely related to the GL functions they represent. This allows the user to use transformation matrix manipulations within an object description file much like he would in application code. Table A.6 lists the current Xformtokens.

Table A.6 Xformtoken Subclasses

A_Scaletok	A_Translatetok	A_Rotatetok	Scaletok
Translatetok	Rotatetok	Pushmatrix	Popmatrix
Loadunit	Loadmatrix	Multmatrix	

Othertokens

The Othertoken class serves as an umbrella for those classes that provide auxiliary functions to NPSDL. These tokens have varying, normally simple, formats. The Othertoken class does not provide any services to it's subclasses. Currently, none of the Othertokens implement the `define()` or `display()` methods. They inherit the default implementations from class Token. Thus their main purpose is to hold information that is used for reading and writing or by other Tokens and NPSobjects. Table A.7 lists the current Othertoken subclasses.

Table A.7 Othertoken Subclasses

Name	Origin	Comment	Lcomment

Note: *The following three sections are essentially the same as the corresponding sections in the main thesis. They are presented here for completeness.*

The NPSCL Run Time Type System

NPSCL provides several support systems that NPSGDL uses extensively. The first is a simple run-time typing system. NPSGDL requires run time type information so that downcasts from a base class pointer to a derived class pointer can be validated. The situations where this is necessary are normally either in the NPSobject implementation or in application code. Because the NPSobject maintains a list of Token pointers, operations like identifying all the Drawtokens requires run time type information. The type system provided by NPSCL is simple and relatively easy to use. The type system is non-intrusive meaning that not all classes must participate, although there is little reason not to include all classes. One thing the type system does not currently support is the ability to query the type system with strings. It might be useful to ask an NPSobject for a list of all the Drawtokens it contains. This operation is not supported because the system has no general association between a typename and the class that contains the type information. Still this type system is useful and effective. It imposes little space overhead on client classes and very little performance overhead by using inline methods where possible. Also since run-time type inquiries are the exception rather than the norm, the system is not a factor in most performance studies conducted.

Basically each class has a public static data member that contains a string identifier for the class and a list of immediate base classes for the class. The class that contains this information is the Type_info class. Methods and macro support allow the user to query the type of a class, determine if a down-cast is safe, compare for type equality and other helpful functions. The user interface to the type system is primarily macros which automate the Type_info class method calls. Alternatively the user can directly access and use a classes Type_info member as it is public. The macro interface relies on the fact that all Type_info data members share the same name and that each class defines required methods to access it's Type_info member. To enforce this, a macro, DeclareType_info, is placed in the public section of every participating class. Similarly, a macro, ImplementType_info(CLASS, NAME, LIST), automates the initialization of the type system with a class's type information.

Using the run time type system is simple. Figure A.10 shows an example of the typical operations that a user of NPSGDL might use. It is very easy to abuse a system like this. In general, designs that rely on virtual function dispatch to get type specific behavior are better than using run time type information and setting up conditionals.

Note: *Refptrs can not be cast directly by the type system. This is because if the cast is valid, the statement 'return (T*) p' is executed. If 'p' is a Refptr you are trying to cast an object which is uncool. When using Refptrs, first convert them to a normal pointer, then test for the down-cast. Exercise caution when mixing normal and Refptrs as noted in the section on reference counting.*

```
// use a base pointer to access a derived object
Token *bp = new Defmaterial;
// test for explicit type
if(ptr_type_info(bp) == static_type_info(Defmaterial))
cout << "This guy is a Defmaterial!" ;
// let's test for valid down-casts
Deftoken *defp = ptr_cast(Deftoken,bp);
Defmaterial *dp = ptr_cast(Defmaterial,bp);
// Both defp and dp should be non-null and valid
Rotatetok *rp = ptr_cast(Rotatetok,bp);
// rp should be null - invalid cast

// We can inquire about an objects heirarchy too
if(ptr_isA(Deftoken,bp))
// add to Deftoken list
```

Figure A.10 Run Time Type System Use

In addition to the string based type system described, NPSCL provides a simple object identification capability based on integers through the Object_id class. The Object_id class provides each client with a unique integer starting at 100. Using this system each class instance is given a unique integer identity that can be used for more refined identity testing. The Object_id class does not allow write access to it's id value. This prevents id sharing or copying. These integer identifiers are used by NPSGDL for various Silicon Graphics, Inc. GL functions, like `lmdef()`, among other things.

Reference Counting System

Another system extensively used by NPSGDL is a simple reference counting garbage collection system. In order to save space and improve efficiency, many token objects are shared between each other and NPSobjects. For example, consider two NPSobjects. Each is associated with a description file that defines a "gold" Defmaterial. We only want one copy of the Defmaterial but each NPSobject must contain a copy in the event that the

NPSobject must write itself to a file. In this case, the first Defmaterial read would allocate space and insert a pointer in a table. The second NPSobject would get a pointer to the first instance rather than a new one when the Defmaterial is read. To support this with a minimum of developer effort, NPSCL provides a simple reference counting system using smart pointer objects. This system is based on the examples in [15] and are similar to the "letter-envelope idiom" in [16]. Basically every class in the NPSGDL system contains a reference count member and methods to increment and decrement the count. This member is introduced by deriving all classes from class Countable. Class Countable is defined in the NPSCL file gencount.H. If an object's reference count decrements to zero, then it is destroyed and its memory is returned to the memory system. The management of the reference counts is the responsibility of a friend class that encapsulates a pointer to the referenced class. This class is generic and uses templates to provide type safety. The pointer class, called a Refptr in NPSCL, overloads operators to behave as a normal pointer with the addition of adjusting reference counts as pointers are assigned, copied and destroyed.

There are several advantages and disadvantages to using this system. Many relating to garbage collection in general:

Advantages

1. System is simple and easy to use.
2. Frees developer from many memory management chores.
3. System is non-intrusive. It can be used or not used as desired.

Disadvantages:

1. System imposes overhead on pointer manipulation. Little overhead is imposed for pointer use or access.
2. User can break system by mixing real pointers and Refptrs.
3. System doesn't detect circular references that might result in unrecoverable memory.

Despite these disadvantages, this system is used in NPSGDL to great success. In normal use, the circular reference problem is not encountered and the overhead is only noticed at non-critical times such as object creation/initialization. Other phases of an object's use normally involves accessing the object pointed at and this operation has little to no overhead due to the use of inline methods.

The primary limitation to using this garbage collection system is placed on the developer. The developer must ensure that real pointers are not mixed with the smart pointers across scopes. This is to prevent the system from deallocating an object still referenced by a real pointer. For example, it would be an error to create a dynamically allocated object within a function, assign the object to a local Refptr and then return the object using a real pointer. The error is that the local Refptr is destroyed on function exit; the refptr will in turn destroy the object since it is the only Refptr referring to it. In general it is not a problem to mix Refptrs and real pointers as long as the real pointer has a smaller scope than the corresponding Refptr. This is not a problem for the typical user as all object management is taken care of within NPSGDL. It is a consideration for developers of tools and those managing custom collections of NPSGDL tokens.

Another consideration is destruction of objects accessed by Refptrs. Refptrs can not be used as arguments to the delete operator. Consider a dynamically allocated object attached to a Refptr named rp. It would be a compiler error to say "`delete rp;`" because delete requires a pointer, not an object, and the Refptr is not converted to a normal pointer in this case. Likewise, deleting the object via a real pointer would be wrong. This would break the reference system because the Refptr would try to delete the object a second time when the Refptr is destroyed, besides the fact that the decrement of the object's reference count would probably fail because the object has been destroyed. To get the desired effect, simply reassign the Refptr thereby decrementing the object's reference count to zero and destroying it. Since NULL is a valid pointer value we can simply say "`rp = 0;`". Zero is preferred over NULL because some NULL definitions result in type mismatches.

Object Sharing

NPSGDL tries to take advantage of as much object sharing as possible. Object sharing serves to reduce memory requirements, reduce data duplications, and provide greater consistency. In general the only objects that are shared within NPSGDL are Deftokens. As mentioned before, each Deftoken class provides a table of pointers to it's instances. These tables are used by clients to access previously defined data. The tables and the classes themselves do not allow duplicates. Since the tables are indexed by token name this means that two tokens of the same type can't share the same name and remain unique. This is important to remember. Let's say two separate object description files define a Defmaterial token named "Brass" but each definition is different. The first file read into the system (typically through an NPSobject) will create a Defmaterial object named "Brass" and add it to the Defmaterial table. When the second file is read (to a second NPSobject), NPSGDL will give the NPSobject a pointer to the first "Brass" object rather than create a second one. Thus if we write the second NPSobject back to a file the "Brass" definition we will get will be the first one. The second is not maintained by NPSGDL. When there is a duplicate clash NPSGDL issues a warning. *Writing objects that have duplicate token clashes can result in lost information!*

Another important point about shared objects is to ensure that the reference system is maintained when using them. The token table in the Deftoken classes are publicly accessible. Developers and users are free to request pointers to tokens by name from the tables. When doing this, it is generally better to use Refptrs to ensure the reference system is maintained. If you use a real pointer to access a token being shared by others you risk the chance that the token will be destroyed by the reference system while you are using it. Also you must ensure that you don't destroy the token via the real pointer as this would invalidate any other Refptrs/pointers accessing the token. If real pointers are used only for short, local scopes there is normally no problem, but the responsibility is the developers.

Having all instances of a class of Deftokens in a central table is very handy. It makes global changes very easy to do. Most Deftokens are used by corresponding Settokens. In most cases the Settokens maintain pointers (Refptr) to a Deftoken. For instance, all Setmaterial tokens named "Brass" refer to the single Defmaterial token named "Brass". Thus if we want to change the characteristics of the "Brass" material, all we have to do is get a pointer to it from the Defmaterial table, make the change, and re-define the token. The next display cycle should show the change through each Setmaterial referencing "Brass". This example is shown in Figure A.11 .

```
#include "Defmaterial.H"

void change_Defmaterial_shiny(string& name, float value) {
// make a global change to a Defmaterial
// get a pointer to the Defmaterial
Refptr(Defmaterial) dp = Defmaterial::table->lookup(name);
if(dp){ // got a valid pointer
  Array(float) dproperties(MATSIZE) = dp->get_properties();
  dproperties[SHINY] = value; // change the shininess
  dp->set_properties(dproperties);
  dp->define(); // redefine for gl
}

dp = 0; // assign to NULL to force dereference
        // this would also happen at end of scope
// array is destroyed at end of if() scope
// NOTE: because the scope of this example is limited we
// could have used a real pointer (Defmaterial *dp)
}
```

Figure A.11 Object Sharing Example

There are many tokens besides Deftokens that could be shared. The Setmaterial "Brass" token in the example is likely duplicated around NPSGDL objects. While there might be some memory space advantages to preventing duplicates in other tokens, the support structures(tables) and algorithms required to implement this was deemed too expensive.

The Persistence Model

A primary requirement for NPSGDL is the ability to store and retrieve object definitions to/from secondary storage. Since C++ does not provide a standard persistence mechanism, one was designed into NPSGDL. The persistence system was modeled after several different systems, in particular the "virtual constructor" methods outlined in [16].

The primary responsibility for storage and retrieval is distributed among the NPSGDL token classes. The most derived class controls most of the process. Each token implements three methods, the read_from, store_on and creator methods as well as a special "reader" constructor called with an input stream. Each takes as input either an input or output stream.

The read_from method expects the stream pointer to be located immediately after the typename of the token and reads all fields on the stream applicable to itself until either an ending flag is encountered, as in Deftokens with a closed format, or a certain number of lines have been read, as with Defpolys which has an open format.

The store_on method stores the token typename and data values in the correct format on a specified stream. This method has a parameter that tells the token whether it should output its typename with its data. Using this parameter, derived classes can have super classes output their data without inserting extraneous typenames on the stream. The only typename that should be put on the stream is the most derived name.

The creator method is a static method. This is important because it does not work on a per object basis. The creator method's function is to allocate a new instance of a token and initialize it from an input stream using the "reader" constructor. The creator then returns a pointer to the new token to the caller. The creator method acts as the virtual constructor as explained below.

Using these methods, each token provides facilities to read, write and initialize itself from streams. Still there must be some object or process in overall control of all this. All of the typenames currently valid in the system must be known in order to recognize them on an input stream. Also, once a typename is recognized, there must be a way of telling the correct token to initialize itself from the stream. This responsibility is delegated to the Token base class. The Token class contains a static table of Tokeninfo objects called the tokentable. A Tokeninfo object associates a token's typename and the address of its creator method. The Token class provides a public static method called read_token() that will process an input stream using the following general algorithm:

1. Read a string, assume it's a token identifier
2. Look up the identifier in the tokentable.
3. If the identifier is valid then get the tokeninfo object for that token, else issue an error and return.
4. Call the identified token's creator method, via the tokeninfo object, passing the input stream in.
5. The creator will return a pointer to a valid token initialized from the stream. Return the token pointer to the caller.

The read_token method is called repeatedly by the NPSobject::read_from method until the end of file is reached.

An important consideration in the design of NPSGDL was the initialization of the tokentable. One of the primary design goals was to be able to add new tokens to the language system with minimum effect on other modules. The ideal being providing a header (.h) and implementation (.C) file for the new token(s) and adding the object module to the library archive. This ideal is very close to being met. The tokentable is initialized dynamically using a special constructor in each token and a special instantiation of each token.

The special constructor is one that takes as its sole argument a Tokeninfo object. As mentioned above, the Tokeninfo object is the object placed into the tokentable. The base class Token's special constructor places the Tokeninfo object into the table. All derived tokens simple call their base class's special constructor passing the Tokeninfo object along to the root of the hierarchy.

In order for this system to work, this constructor must be used. Thus in each tokens implementation file, a single static object of the particular token type is instantiated using the special constructor. The C++ language guarantees that global static objects will be constructed before main() is entered. So all of the static objects used for tokentable initialization are constructed prior to main() ensuring that the tokentable is properly initialized automatically at run-time. There is no need for user initialization. The Token class's special constructor ensures that the tokentable is instantiated before it is used.

Since the special constructor is normally the first constructor that will be called for a class, it is ideal for initializing any static data structures or default tokens. This is exactly what is done in the Deftoken classes. Each Deftoken instantiates it's table within the special constructor, ensuring that the table is properly initialized before any tokens are added to it. Also the special constructor creates a default token, named "default" when named, that is placed into the table. The default token is used by client Settokens when a Deftoken name is not recognized.

This system for dynamic initialization works very well. There is one problem though, current linker technology does not support this model well. Typical NPSGDL users use only the NPSobject class and don't directly refer to the token subclasses. Since the NPSobject classes deals with tokens in the abstract through Token pointers, it does not refer to derived tokens directly either. Thus there are normally no references to derived token modules for the linker to resolve. The result is that the linker does not include the modules for derived classes, the special instances of derived tokens are not constructed, and the tokentable is not initialized properly.

The temporary remedy was to fall back to the more traditional method of having an object or module that ensures that each module is linked in. Instead of having an external class manage the tokentable initialization, there is an external object, called a TokenRegistrar, whose constructor calls a static method named register_token() provided by each token class. This method does nothing. It is used solely to generate an unresolved reference for the linker. The TokenRegistrar is instantiated in the Token module which is always linked in. The advantage to this approach is that the registration functionality is easily removed without affecting anything.

There is another consequence to using the registration method. NPSobjects only know about the class Token. The actual derived tokens used are instantiated at run-time and accessed via dynamically bound calls. Thus there is no way to know which tokens will be needed by any set of definition files. Therefore all token modules must be linked to the application. This results in large applications with possibly a lot of unused code. The alternative is to preprocess definition files and only link the modules needed. However, this is very limiting and establishes an application dependance. The definition files can't be modified without preprocessing again. This method of linking (selective) is possible with a minor modification to the NPSGDL system, outlined below.

This should only be done by the primary developer responsible for maintenance of NPSGDL!

To allow selective linking of NPSGDL modules you need to modify the Token.C module and recompile the library. Comment out or remove the statement "`static Token_registrar registrar;`" on line 22 of Token.C. The Token_registrar class constructor contains calls to the register_token() method of all NPSGDL tokens. After this change is done it is up to the user to ensure that the proper token object modules are linked to an application.

GDL Macro Maker (GDLMM)

The utility GDLMM, GDL macro maker, is provided to assist in determining which token modules are needed by an application. GDLMM works in conjunction with your makefile to automate the linking process. GDLMM will read the NPSGDL object description files provided to it and generate linking information for the token modules needed for those object files. This linking information is inserted into your makefile in the form of a macro that you use in your application compilation statement.

The syntax of GDLMM is:

gdlmm [-o exename] [-f makefilename] GDL_filename [...]

-f: The name of the makefile to modify. Defaults to "Makefile"

-o: The name of the application being processed. Defaults to "". This name is used to name the macro generated by GDLMM.

The GDLMM algorithm is fairly simple. First the file "gdlmodule.info" is read. This file contains associations between token names and module pathnames. This info file must be correct and up to date for the generated linking macros to be correct.

Second, GDLMM reads each GDL object file listed on the command line and marks each token module that needs to be linked.

Third, GDLMM processes the makefile adding or modifying the macro definition for the application being processed. It does this by scanning the makefile specified for one of two strings at the beginning of a line: "exename_GDLMODULES= "or "#macroend" where exename is the name provided after the -o switch. The string "#macroend should be placed at the end of your makefile's macro definition area. GDLMM uses this string to stop processing the makefile and to mark the location that a new macro should be placed before. If your makefile does not contain either of the above strings, GDLMM will not output any new macro definition.

Once GDLMM has processed an application's GDL object files and modified the makefile, all you need to do is include the appropriate "_GDLMODULES" macro in your compilation line.

If you are using this system of linking and want all of the token modules linked in as in the normal system, simple define the link macro to link to the "tokreg.o" module. An example makefile with GDLMM generated macros is shown in Figure A.12 .

This method of linking individual modules bypasses some of the normal object archive processing. In particular all of the NPSGDL object code modules need to be available to the user. Also all of the object modules should be compiled the same. Strange things might happen if the user links to modules with mixed debugging or optimization modes. In general, If this system is used the NPSGDL object modules used for linking should be the same as those used to build the normal, optimized library (no debug info).

```
# main makefile for NPSGDL programs

# this forces the linkage of all GDL modules
showobj_GDLmodules= /n/gravy2/work/wilson/thesis/NPSGDL/
Tokreg.o

robot_GDLmodules= /n/gravy2/work/wilson/thesis/NPSGDL/
A_Rotatetok.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Defpcamera.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Loadunit.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Setmaterial.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Translatetok.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Defcone.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Defvariable.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Defcylinder.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Pushmatrix.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Popmatrix.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Defsphere.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Comment.o \
/n/gravy2/work/wilson/thesis/NPSGDL/Lcomment.o

#macroend

.SUFFIXES: .C .o

# implicit compile rules
.C.o:;  CC -c $(DBGFLAGS) $(INCLUDE) $*.C
.c.o:;  cc -c $(NDBGFLAGS) $(INCLUDE) $*.c

showobj: showobj.o ../libDNPSGDL.a ../../NPSCL/libNPSCL.a
        CC -o showobj showobj.o $(showobj_GDLmodules) $(DBGFLAGS)
$(INCLUDE) $(LIBDIRS) $(DNPSGDL_LIB) $(NPSCL_LIB) $(GL_LIB)
$(IMAGE_LIB)

robot: robot.o ../libDNPSGDL.a ../../NPSCL/libNPSCL.a
        CC -o robot robot.o $(robot_GDLmodules) $(DBGFLAGS)
$(INCLUDE) $(LIBDIRS) $(DNPSGDL_LIB) $(NPSCL_LIB) $(GL_LIB)
$(IMAGE_LIB)

# DO NOT DELETE THIS LINE -- make depend depends on it.
```

Figure A.12 GDLMM macro example

Extending NPSGDL

One of the primary requirements in the design of NPSGDL was easy extension. This is one of the prime reasons the object-oriented paradigm was chosen. This section will illustrate, in detail, the process of adding a new language token to the NPSGDL system. This section will provide the knowledge needed to add and/or maintain NPSGDL token classes. It also discusses, in detail, the responsibilities of each method in a token class and the considerations when implementing them.

The example extension will consist of a pair of cooperative tokens. These tokens were actually added to NPSGDL for this example. The two tokens are the Defsound and Setsound token. Here we will look at the Defsound token in depth and cover the Setsound token later.

The first step is to define the role of the token within NPSGDL. NPSGDL was designed to support application development in the graphics and video laboratory. Recent hardware and software additions provide support for producing and controlling sound effects within an application. By providing support for sound effects within NPSGDL we offer an application independent method for managing sounds.

There are many issues that need to be addressed concerning the use of sound. Reference LD present many of these issues and a solution currently used in the laboratory. Essentially, sound commands are sent via a serial to MIDI line to a digital sampler/ sequencer keyboard. The sound command is a hexadecimal value that corresponds to a particular sound on the keyboard. These sounds range from short transient sounds like gunshots to continuous sounds like vehicle movement. A simple application programming interface (API) is provided to control and produce sounds in the manner described above.

How NPSGDL will support sounds needs to be decided. Users typically use NPSGDL on an object basis. That is, each major object in an application is described completely within a single description file with possible sharing of definitions (Deftokens). Thus the user thinks of a description file as an object with the understanding that the object will typically be displayed each display cycle. This has important ramifications concerning sounds. There are few sounds associated with an individual object that a user will want to be played once per cycle. This would happen because we are going to use the Deftoken/ Settoken model and Settokens are all displayed each cycle (a different model could be used for sounds with a little more effort). Normally the user will want sounds tied to specific object events. A separate class structure would probably be used to bring the graphical and behavioral nature of an object together using NPSGDL and possibly other systems. This does not mean that sounds won't work well within NPSGDL. Remember that

NPSGDL also supports a component based style of programming. Each of the token classes can be used on it's own under user control rather than NPSGDL control. Using this style we could have an NPSGDL description file that only contains Defsound definitions. This file would be read into an NPSobject and that NPSobject defined early in an application's execution. Then the user would be free to instantiate Setsound objects as needed and simply display the Setsound to play it. Using NPSGDL gives us these advantages:

1. The ability to easily associate names to sound codes in an application independent manner through Defsounds.
2. The ability to maintain numerous sets of sound definitions that can be selected at run time or changed without application recompilation.
3. The ability to select and play sounds by name within an application through Setsound tokens.
4. A sound management model that is easily changed or extended. Sounds might correctly be an entirely new category of tokens.

We will design our Defsound/Setsound pair as follows:

1. The Defsound token will be derived from the Deftoken class. This is for two reasons: First, semantically the Defsound token is associating and storing data, the hex value, for later use by other tokens. All Deftokens do this. Second, the Deftoken class will handle the reading, writing, and management of the token name for us.
2. The Defsound token will associate a string name to a hexadecimal sound value. All normal Deftoken methods will be implemented except possibly `define()` because there is nothing that needs to be system defined.
3. The format of a Defsound will be:

   defsound name hexvalue

4. The Setsound token will be derived from the Settoken class. This also fits the semantics of Settokens. Settokens normally utilize the data in Deftokens to activate or change a system state. The only anomaly of using Settokens is having to use the `display()` method, a semantic name mismatch, to produce the sound. The `display()` method must be used to ensure that a Setsound can be used polymorphically with other tokens. The `display()` method of a Settoken effects its main behavior.
5. The Setsound token will have a name that matches a previously read or

instantiated Defsound name. The Setsound token will implement all the normal Settoken methods. `Setsound::display()` will issue the sound play command using the hex value of the corresponding Defsound.

6. The format of a Setsound will be:

   setsound name

   Note that based on the previous discussion Setsounds will probably be used directly in applications rather than within description files.

Now we are ready to start implementing the tokens. We will start with the Defsound class. The first thing to do is set up any support classes that are needed. Defsound is a Deftoken and all Deftokens maintain a table of class instances. This table will be indexed by name (strings) and contain Refptrs to Defsounds. This requires two support classes, Refptr(Defsound) and HashTable(string,Defsound). Both are instantiations of NPSCL template classes. Each of these support classes will have it's own header and source file. The contents of these files is shown in Figure A.13 .

```
/*******************************************
File Dsoundptr.H
********************************************/
#ifndef DefsoundPTR_H
#define DefsoundPTR_H

#include "Defsound.H"
#include "gencount.H"

DeclareRefptr(Defsound);
#endif
/*******************************************
FILE: Dsoundtab.H
********************************************/
#ifndef Defsound_TABLE_H
#define Defsound_TABLE_H

#include "Dsoundptr.H"
#include "gentable.H"
#include "String.H"

DeclareHashTable(string,Refptr_Defsound);
#endif
/*******************************************
FILE: Dsoundtab.C
********************************************/
#include "Dsoundtab.H"

ImplementHashTable(string,Refptr_Defsound);
ImplementHashTable2(string,Refptr_Defsound);
```

Figure A.13 Defsound Support Files

With the support classes implemented, we can start on the Defsound class. The Defsound class is publicly derived from Deftoken. Defsound will inherit all the methods defined by Deftoken and Token. This includes the standard methods required for all tokens as well as special functions unique to Deftokens, like name management methods. Most of the methods that Defsound inherits are virtual functions. This gives the developer the chance to redefine the default Token or Deftoken methods for the Defsound.

The first thing to do is declare the class in its own header file. In general, most of the token classes have similar structures. This makes token creation easy because in many cases you can find a similar class, copy its files and change the names. In fact the Defsound/Setsound class is similar to the Defcolor/Setcolor class to they were used as templates. The header file for Defsound is shown in Figure A.14 .

```
/*********************************************
 FILE: Defsound.H
 PROGRAMMER: Kalin Wilson
 LANGUAGE: C++
 ENVIRONMENT: SGI Iris
 DESCRIPTION: This file contains the declaration of the Defsound
token for NPSGDL.
 *********************************************/
#ifndef _Defsound_H
#define _Defsound_H

#include "Deftoken.H"

// forward declarations to preclude including headers
class string;
class HashTable_stringRefptr_Defsound;

class Defsound : public Deftoken {
  friend class Setsound;  // so Setsound can directly access data
public:
  DeclareType_info;          // insert the type system members
  Defsound();                // default constructor soundval=0;
  Defsound(const Defsound&); // Copy constructor.
  Defsound(Tokeninfo);       // tokentable initializer
  Defsound(istream&);        // reader constructor
                           // construct from string and sound value
  Defsound(const string&, int);
  ~Defsound();               // destructor

  void operator=(const Defsound&);  // allow assignment of tokens
                                    // compare names and soundval
virtual bool operator==(const Token&) const;
                                    // order on soundval
  virtual bool operator<(const Token&) const;
                                 // read name and soundval from stream
  virtual void read_from(istream&);
                                    // write a Defsound to a stream
  virtual void store_on(ostream&, bool savetype = TRUE) const;
  virtual Token *clone() const;       // make a distinct copy

  /************** special funcs *************/
                                      // change name and update table
 virtual void set_name(const string&);
  int get_soundvalue() const;           // retrieve the sound value
  void set_soundvalue(const int);       // define the sound value

  ///////////  static functions //////////////
                 // create a Defsound from stream resolving duplicates
  static Token *creator(istream&);

  static void register_token();        // NOOP used to force linking
                                 // check for duplicate on construction
  static bool duplicate_error();

  // public data member
                                       // Table containing instances
  static HashTable_stringRefptr_Defsound *table;
                                      // method to print entire table
  static void dump_table_on(ostream&);
private:
                 // indicates a duplicate was constructed
  static bool construction_error;
  int soundval;
};

#endif
```

Figure A.14 Defsound Header File

Note the use of the `DeclareType_info` macro in the public section of the class. Also the Hashtable(string,Refptr_Defsound) pointer declaration. The macro is used to insert the required members to support the run time type system. The pointer is static and bound to a dynamically allocated object by the constructors.

Now let's look at the implementation of the Defsound class. As a matter of style, the first thing I do, following header inclusion, is to instantiate all static member data. Second I add the class's type information to the type system. This is done by first declaring a static array of const Type_info pointers initialized to point at base class Type_info members. Only immediate base classes are needed so only one will be listed unless multiple inheritance is used. The array must be terminated by a null word. The "`ImplementType_info`" macro is provided in `Typeinfo.H` to help initialize the type system. It properly formats the instantiation of the static data member, `Type_info class_desc`, included in every token class. The arguments to the macro are: class name, class identifier[1], and pointer to array of pointers. To get a match between the string read from a stream and the one stored in the Token::tokentable.

Following the type system initialization, I instantiate a static object that uses the tokentable initializer constructor. Since this object is global/static it will be constructed prior to main(). When it is constructed it will insert its information into the Token::tokentable to allow virtual construction from streams.

Next we need to define the constructors for a Defsound. All token classes must provide at least four different constructors for the system to work. These are: a default constructor, a copy constructor, a tokentable initializer constructor, and a reader constructor.

The default constructor must initialize the Defsound to a valid default state. It must also ensure that the proper base class constructors are called. The default constructor is used to support array of classes, a convenient syntax, and to support good programming

1. **Important**: *The class identifier must be all lower case!. To provide case insensitivity in the token formats, the Token::read_token() method converts all potential identifiers to lower case.*

practices. The Defsound default constructor is shown in Figure A.15 . Note: *The "TRACER" statements are conditionally compiled debugging statements. For more information see the NPSCL reference manual.*

```
Defsound::Defsound() : Deftoken()
{
 TRACER("Defsound::Defsound()");

 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }
 construction_error = FALSE;
 soundval = 0;
 // The default constructor does not insert tokens into the table
}
```

Figure A.15 Defsound Default Constructor

The copy constructor is used to initialize a new Defsound from an existing one. It is important to provide a copy constructor because it is used to initialize temporary objects, function arguments and other things that normally go unnoticed. Notice that the Defsound constructors call Deftoken constructors first. You should use base class constructors to initialize base class members. This keeps responsibility localized and consistent. It is also required if the base class data is private. The copy constructor for Defsounds is shown in Figure A.16 .

```
Defsound::Defsound(const Defsound& d) : Deftoken(d), soundval(d.soundval)
{
 TRACER("Defsound::Defsound(const Defsound& d) : name(d.name)");

 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }
 construction_error = FALSE;

 // this produces a duplicate so it is not added to the table
}
```

Figure A.16 Defsound Copy Constructor

The tokentable initializer constructor is used to add a tokens information into the Token::tokentable. As discussed above, this table is used to implement virtual construction of tokens from a stream. The tokentable initializer constructor does double duty in many tokens.

Since this constructor is usually the first called (for the class) during execution, it is used to initialize the default token and add it to the tokens table. The Defsound tokentable initializer constructor is shown in Figure A.17 .

```
Defsound::Defsound(Tokeninfo ti) : Deftoken(ti)
{
 TRACER("Defsound::Defsound(Tokeninfo ti) : Token(ti)");

 // This will normally build the table
 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }
 construction_error = FALSE;
 // define default token and add to table
 soundval = 0;
 table->add(name, this);

}
```

Figure A.17 Defsound Tokentable Initializer

The last constructor required for all tokens is the reader constructor. The reader constructor is responsible for initializing a token from a stream, usually a file stream. The reader constructor begins with the assumption that the token identifier has been read. The identifier was read by the `Token::read_token()` method and the reader constructor is called from the tokens `creator()` method. The reader constructor is responsible for parsing the stream for it's information and providing reasonable defaults and error recovery. In the case of Deftokens, if a duplicate is constructed then the caller should be notified via a flag. The reader constructor calls the `read_from()` method to do the actual reading. This keeps the code simple and localized. Figure A.18 shows the Defsound reader constructor.

```
Defsound::Defsound(istream& is) : Deftoken()
{
 TRACER("Defsound::Defsound(istream& is) READER");
 // This is the reader constructor for a Defsound

 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }

 // reset the error flag
 construction_error = FALSE;

 read_from(is); // init from stream and return
}
```

Figure A.18 Defsound Reader Constructor

The last constructor provided by the Defsound class allows users to define a Defsound explicitly. This constructor is not used by the NPSGDL system. But since all tokens should be somewhat stand-alone, constructors to build tokens directly should be provided. This constructor uses Deftoken constructor to initialize the name and checks for duplicates. It is shown in Figure A.19 .

```
Defsound::Defsound(const string& n, int s): Deftoken(n), soundval(s)
{
 TRACER("Defsound::Defsound(const string& n, const vertex& c)");
 // add this new token to the table if it's not already there

 construction_error = FALSE;
 if(!table->includes(name))
 table->add(name,this);
 else
 construction_error = TRUE; // this is a duplicate
}
```

Figure A.19 Defsound Constructor

The destructor for the Defsound class is empty. Since dynamic memory is not used by Defsounds, there is no need to explicitly destroy member objects. It is good practice to define a destructor in any case as a simple check that everything that needs to be implemented has been. The Token destructor is virtual so all derived token destructors are too. This ensures that if delete is called on a base pointer, the entire object referenced by the pointer is destroyed properly.

The last method involved directly in token construction is the `creator()` method. The `creator()` method is a static method used to implement a kind of "virtual" constructor. The `creator()` method must be static so that it can be called via a pointer without a class object. The `creator()` method is responsible for allocating memory for a new object, calling the proper constructor and resolving duplicates if necessary. The `creator()` method is normally only used by the persistence system when reading description files. The `creator()` method for Defsound is shown in Figure A.20 .

```
Token *Defsound::creator(istream& is)
{
 TRACER("Token *Defsound::creator(istream& is)");

 Defsound *newtoken = new Defsound(is);

 if(!newtoken)
 Warning("Memory allocation error during creation of Defsound");
 else if(construction_error){
 // the Defsound just built is a duplicate
 string dname = newtoken->get_name();
 delete newtoken; // get rid of the dup
 newtoken = table->lookup(dname); // get a pointer to the named Defsound
 }
 return newtoken;

}
```

Figure A.20 Defsound Creator Method

Now let's look at the reading and writing methods for the Defsound. The `read_from()` method parses an input stream and extracts the tokens data. It also assumes the tokens identifier has already been read. The normal algorithm for a `read_from()` method is:

1. Have any base classes read their data.
2. Read additional data this token requires.

This is illustrated in Figure A.21 .

```
void Defsound::read_from(istream& is)
{
 string w;

 // have the Deftoken read the name
 Deftoken::read_from(is);

 // read the sound code after the name
 if(!(is >> soundval)) {
 w = "Unexpected error while reading sound value for Defsound " & name;
 Warning(this,w);
 }

 construction_error = FALSE;
 // store the new sound definition in the table
 if(table == 0){
 w = "Defsound table has not been allocated prior to Defsound token ini\
tialization";
 Warning(this, w);
 }
 else if(table->includes(name)) {
 w = "Defsound " & name & " already defined! Ignoring redefinition.";
 Warning(w);
 construction_error = TRUE;
 }
 else {
 // This is a new definition, just store it in the table
 table->add(name, this);
 }
}
```

Figure A.21 Defsound Read_from Method

The method for writing a token definition to a stream is `store_on()`. The `store_on()` method is usually simple. It has an extra parameter that is used to tell the method to output the token identifier or not. This is important because a derived tokens `store_on()` method will call base `store_on()` methods but we only want the derived token to put a token identifier on the stream. Thus, the algorithm for writing a token is:

1. Write identifier is necessary.
2. Have base classes write their data. Base class methods should be called in the same order they appear in the class declaration as this is the order they will be constructed.
3. Write token specific data

The store_on method for the Defsound class is shown in Figure A.22 .

```
void Defsound::store_on(ostream& os, bool savetype) const
{
 TRACER("void Defsound::store_on(ostream& os, bool savetype)");

 if(savetype) {
 os << class_desc.name();
 }
 // write the name without the Deftoken id
 Deftoken::store_on(os, FALSE);

 // output the sound code definition
 os << soundval << endl;

}
```

Figure A.22 Defsound Store_on Method

The above code makes up the bulk of what is needed to implement the Defsound token class. The only other methods that should be defined are the overloaded operators, the `clone()` method and the `register_token()` method. The `register_token()` method is empty. It is only used by the Token_registrar object to force modules to link to an application. The `clone()` method is provided to allow users to get a distinct copy of an object when managing tokens through pointers. Both of these methods require no change if you imported another tokens source, beyond changing the token names.

The overloaded operators allow the user to use a simple, intuitive syntax when using tokens.You should overload `Operator=()` to allow users to assign tokens to each other. Also, if appropriate for the token, `operator==()` and `operator<()` should be defined for comparisons. The implementations of these methods is straightforward and shown in Figure A.23 . Note that the Defsound class does not implement a `define()` method. Since there is nothing that needs to be provided to the graphics or any other system, there is no reason to override the default define method which does nothing.

```
Token *Defsound::clone() const
{
 TRACER("Token *Defsound::clone() const");

 return (Token *) new Defsound(*this);
}


void Defsound::operator=(const Defsound& d)
{
 TRACER("void Defsound::operator=(const Defsound& d)");

 if(this == &d)
 return;

 Deftoken::operator=(d);
 soundval = d.soundval;
}

bool Defsound::operator==(const Token& t) const
{
 TRACER("bool Defsound::operator==(const Token& t) const");

 if(ref_isA(Defsound,t)){
 // got a valid type so it's OK to cast-down

 Defsound& temp = (Defsound&) t;
 if(!Deftoken::operator==(temp))
 return FALSE;
 if(soundval != temp.soundval)
 return FALSE;

 return TRUE;
 }
 else
 return FALSE;
}

bool Defsound::operator<(const Token& t) const
{
 if(ref_isA(Defsound,t)){
 Defsound &temp = (Defsound&) t;
 return soundval < temp.soundval;
 }
 else
 return FALSE;
}
```

Figure A.23 Defsound Miscellaneous Methods

The implementation of the Setsound token is much the same. The same approach is used and the same considerations need to be addressed that were discussed above. The main difference is that we need to define a `display()` method that will implement the run-time behavior of the Setsound. A simple call to the sound routine that plays a sound on the synthesizer is all that is needed. Since the Setsound is a friend to Defsound, we can directly access the sound code that is sent to the sound routines. Directly referencing the

Defsound token ensures that if the user redefines the sound code the change will be seen by all Setsounds that use it. The implementation for the `Setsound::display()` method is shown in Figure A.24 . The complete implementation for the Defsound and Setsound token classes is contained in APPENDIX B: DEFSOUND AND SETSOUND TOKEN IMPLMENTATIONS.

```
void Setsound::display() const
{
 TRACER("void Setsound::display() const");
 if(defptr) {
// the sound system is not fully implemented yet
// soundplay(defptr->soundval);
 }
}
```

Figure A.24 Setsound Display Method

Adding tokens to the libraries

Once the source files for the new tokens are finished, they need to added to the make files so they can be compiled and added to the NPSGDL libraries. There are three subdirectories used to manage the NPSGDL libraries. Each has it's own make file.The "debug" directory makefile compiles using the -g switch for debugging information and builds the library archive "libDNPSGDL.a". The "optimized" directory makefile compiles using the -O switch for optimized code and defines the macro "NDEBUG" which disables assertion checking. It manges the "libNPSGDL.a" archive. This is the library normally used by designers. The last subdirectory, "trace", has a makefile that compiles using the -g debugging switch and defines the macro "DEBUG" which enables function call tracing via the NPSCL debugging classes in "debug.H". This library is useful to NPSGDL maintenance developers who need to monitor NPSGDL activities. As mentioned before, special steps must be taken to initialize the tracing system to prevent errors.

The source code for the NPSGDL system is contained in the NPSGDL directory. Different versions of object files as well as the libraries are kept in the library subdirectories. In order for make to manage the file dependencies properly, file links to the source files are also placed in each library directory.

There are some basic steps that need to be followed to ensure a new token is compiled correctly, added to all the libraries and will be linked to an application when needed. These are shown below. In addition to those mentioned below, there are other make directives available in the main NPSGDL makefile. They allow updates to individual libraries and removal of the file links. See the makefile for details.

1. In the Token_registrar constructor, add a call to the new tokens register_token() method. This is to force linking of the token to make sure it's information is added to the tokentable.
    1. Open the file Tokreg.H.
    2. Add the new token's header file to Tokreg.H.
    3. Add a call to the new token's register_token() method to the Token_registrar constructor.
2. Add the new token's source and object file names to the "SRC" and "OBJ" macros in each makefile of each library directory.
3. From the NPSGDL directory, run "make depend" to update the library file dependency information.
4. From the NPSGDL directory, run "make updatelinks" to update the source file links in each library directory.
5. From the NPSGDL directory, run "make" to rebuild all of the libraries.

Each library has hard links in the NPSGDL/lib directory for application programers.

This method of managing the different versions of the NPSGDL system keeps multiple copies of the object files. This can be useful if the alternate linking scheme discussed above is used. Otherwise they take up space that may be reclaimed after the libraries have been built.

APPENDIX B: DEFSOUND AND SETSOUND TOKEN IMPLMENTATIONS

```
/********************************************
 FILE: Defsound.H
 PROGRAMMER: Kalin Wilson
 LANGUAGE: C++
 ENVIRONMENT: SGI Iris
 DESCRIPTION: This file contains the declaration of the Defsound token for
 NPSGDL.
 ********************************************/

#ifndef _Defsound_H
#define _Defsound_H

#include "Deftoken.H"

// forward declarations to preclude including headers
class string;
class HashTable_stringRefptr_Defsound;

class Defsound : public Deftoken {
 friend class Setsound; // so Setsound can directly access data
public:
 DeclareType_info; // insert the type system members
 Defsound(); // default constructor soundval=0;
 Defsound(const Defsound&); // Copy constructor.
 Defsound(Tokeninfo); // tokentable initializer
 Defsound(istream&); // reader constructor
 Defsound(const string&, int); // construct from string and sound value
 ~Defsound(); // destructor

 void operator=(const Defsound&); // allow assignment of tokens
 virtual bool operator==(const Token&) const; // compare names and
soundval
 virtual bool operator<(const Token&) const; // order on soundval
 virtual void read_from(istream&); // read name and soundval from stream
 // write a Defsound to a stream
 virtual void store_on(ostream&, bool savetype = TRUE) const;
 virtual Token *clone() const; // make a distinct copy

 /************** special funcs *************/
 virtual void set_name(const string&); // change name and update table
 int get_soundvalue() const; // retrieve the sound value
 void set_soundvalue(const int); // define the sound value

 /////////// static functions //////////////
 static Token *creator(istream&); // create a Defsound from stream
 // resolving duplicates
 static void register_token(); // NOOP used to force linking
 static bool duplicate_error(); // check for duplicate on construction
 // public data member
 static HashTable_stringRefptr_Defsound *table; // Table containing
instances
 static void dump_table_on(ostream&); // method to print entire table
private:
 static bool construction_error; // indicates a duplicate was constructed
 int soundval; // the code used to access sounds
};


inline int Defsound::get_soundvalue() const { return soundval;}
inline void Defsound::set_soundvalue(const int s) { soundval = s;}
#endif

/*******************************************
 FILE: Defsound.C
 PROGRAMMER: Kalin Wilson
 LANGUAGE: C++
 ENVIRONMENT: SGI
 DESCRIPTION: This file contains the implementation of the Defsound token
 *******************************************/


#include "Defsound.H"
#include "Dsoundtab.H"
#include "Dsoundptr.H"
#include "debug.H"

// #include "sound.h"
// #include "network.h"

// instantiate the static data members
HashTable_stringRefptr_Defsound *Defsound::table = 0;
bool Defsound::construction_error = FALSE;

// initialize the type info system
static const Type_info *Defsound_b[] = {&Deftoken::class_desc, NULL};
ImplementType_info(Defsound,"defsound",Defsound_b);

// Defsound instance to init tokentable and define default
static Defsound Defsound_init(Tokeninfo(Defsound::class_desc.name(),
&Defsound::creator));


/***********************************************************/
Defsound::Defsound() : Deftoken()
{
 TRACER("Defsound::Defsound()");

 if(table == 0) {
```

```
 table = new HashTable(string,Refptr_Defsound)(111);
 }
 construction_error = FALSE;
 soundval = 0;
 // The default constructor does not insert tokens into the table
}


Defsound::Defsound(const Defsound& d) : Deftoken(d), soundval(d.soundval)
{
 TRACER("Defsound::Defsound(const Defsound& d) : name(d.name)");

 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }
 construction_error = FALSE;

 // this produces a duplicate so it is not added to the table
}

Defsound::Defsound(istream& is) : Deftoken()
{
 TRACER("Defsound::Defsound(istream& is) READER");
 // This is the reader constructor for a Defsound

 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }

 // reset the error flag
 construction_error = FALSE;

 read_from(is); // init from stream and return
}


Defsound::Defsound(Tokeninfo ti) : Deftoken(ti)
{
 TRACER("Defsound::Defsound(Tokeninfo ti) : Token(ti)");

 // This will normally build the table
 if(table == 0) {
 table = new HashTable(string,Refptr_Defsound)(111);
 }
 construction_error = FALSE;
 // define default token and add to table
 soundval = 0;
 table->add(name, this);

}
```

```
Defsound::Defsound(const string& n, int s): Deftoken(n), soundval(s)
{
 TRACER("Defsound::Defsound(const string& n, const vertex& c)");
 // add this new token to the table if it's not already there

 construction_error = FALSE;
 if(!table->includes(name))
 table->add(name,this);
 else
 construction_error = TRUE; // this is a duplicate

}

Defsound::~Defsound()
{
 TRACER("Defsound::~Defsound()");
 // nothing to explicitly destroy
}


Token *Defsound::creator(istream& is)
{
 TRACER("Token *Defsound::creator(istream& is)");

 Defsound *newtoken = new Defsound(is);

 if(!newtoken)
 Warning("Memory allocation error during creation of Defsound");
 else if(construction_error){
 // the Defsound just built is a duplicate
 string dname = newtoken->get_name();
 delete newtoken; // get rid of the dup
 newtoken = table->lookup(dname); // get a pointer to the named Defsound
 }
 return newtoken;

}

void Defsound::operator=(const Defsound& d)
{
 TRACER("void Defsound::operator=(const Defsound& d)");

 if(this == &d)
 return;

 Deftoken::operator=(d);
 soundval = d.soundval;
}

bool Defsound::operator==(const Token& t) const
```

```
{
 TRACER("bool Defsound::operator==(const Token& t) const");

 if(ref_isA(Defsound,t)){
 // got a valid type so it's OK to cast-down

 Defsound& temp = (Defsound&) t;
 if(!Deftoken::operator==(temp))
 return FALSE;
 if(soundval != temp.soundval)
 return FALSE;

 return TRUE;
 }
 else
 return FALSE;
}

bool Defsound::operator<(const Token& t) const
{
 if(ref_isA(Defsound,t)){
 Defsound &temp = (Defsound&) t;
 return soundval < temp.soundval;
 }
 else
 return FALSE;
}

void Defsound::read_from(istream& is)
{
 string w;

 // have the Deftoken read the name
 Deftoken::read_from(is);

 // read the sound code after the name
 if(!(is >> soundval)) {
 w = "Unexpected error while reading sound value for Defsound " & name;
 Warning(this,w);
 }

 construction_error = FALSE;
 // store the new sound definition in the table
 if(table == 0){
 w = "Defsound table has not been allocated prior to Defsound token
initialization";
 Warning(this, w);
 }
 else if(table->includes(name)) {
 w = "Defsound " & name & " already defined! Ignoring redefinition.";
 Warning(w);
 construction_error = TRUE;
 }
 else {
 // This is a new definition, just store it in the table
 table->add(name, this);
 }
}

void Defsound::store_on(ostream& os, bool savetype) const
{
 TRACER("void Defsound::store_on(ostream& os, bool savetype) const");

 if(savetype) {
 os << class_desc.name();
 }
 // write the name without the Deftoken id
 Deftoken::store_on(os, FALSE);

 // output the sound code definition
 os << soundval << endl;

}

Token *Defsound::clone() const
{
 TRACER("Token *Defsound::clone() const");

 return (Token *) new Defsound(*this);
}


void Defsound::register_token()
{
TRACER("void Defsound::register_token() ");
}



void Defsound::dump_table_on(ostream& os)
{
 TRACER("void Defsound::dump_table_on(ostream& os)");
 if(!table)
 return;

 table->goto_first();
 while(table->current_valid()) {
 (table->current_data())->store_on(os);
 table->goto_next();
 }
}
```

```
void Defsound::set_name(const string& n)
{
 // This function overrides the Deftoken function because
 // we need to update the Defsound Table

 table->remove(name);
 name = n;
 table->add(name,this);
}

bool Defsound::duplicate_error() { return construction_error;}

/***************** end of Defsound.C *************************/

// -*- C++ -*-
/********************************************
FILE: Setsound.H
PROGRAMMER: Kalin Wilson
LANGUAGE: C++
ENVIRONMENT: SGI
DESCRIPTION: This file contains the declaration of the Setsound token
********************************************/

#ifndef SETSOUND_H
#define SETSOUND_H

#include "Settoken.H"
#include "Dsoundptr.H"


class Setsound : public Settoken {
public:
 Setsound();
 Setsound(const Setsound&);
 Setsound(Tokeninfo);
 Setsound(istream&);
 virtual ~Setsound();

 void operator=(const Setsound&);
 virtual bool operator==(const Token&) const;
 virtual void read_from(istream&);
 virtual void store_on(ostream&, bool savetype = TRUE) const; // Writer
 virtual Token *clone() const;
 virtual void display() const;
 virtual void set_name(const string&); // override to get new assosiation
 DeclareType_info;
 static Token *creator(istream&);
 static void register_token();
```

```
private:
 Refptr(Defsound) defptr;
};


#endif


/*******************************************
 FILE: Setsound.C
 PROGRAMMER: Kalin Wilson
 LANGUAGE: C++
 ENVIRONMENT: SGI
 DESCRIPTION: This file contains the implementation of the Setsound token
 *******************************************/

#include "Setsound.H"
#include "Defsound.H"
#include "Dsoundptr.H"
#include "Dsoundtab.H"
#include "debug.H"

// init the type system
static const Type_info *Setsound_b[] = {&Settoken::class_desc, NULL};
ImplementType_info(Setsound,"Setsound",Setsound_b);

// init the tokentable
static Setsound Setsound_init(Tokeninfo(Setsound::class_desc.name(),
&Setsound::creator));

/*********************************************************/
Setsound::Setsound() : Settoken()
{
 TRACER("Setsound::Setsound()");
 string w;

 // get the default sound
 if(Defsound::table) {
 defptr = Defsound::table->lookup(name);
 if(!defptr) {
 w = "Defsound " & name & " not found in Defsound::table. Using
'default'.";
 Warning(this, w);
 defptr = Defsound::table->lookup("default");
 if(!defptr) {
Warning("Defsound 'default' not defined in Defsound:table.");
 }

 }
 }
 else {
```

```
 w = "Defsound::table has not been initialized";
 Warning(this, w);
 }

}


Setsound::Setsound(const Setsound& d) : Settoken(d), defptr(d.defptr)
{
 TRACER("Setsound::Setsound(const Setsound& d) : name(d.name)");
}


Setsound::Setsound(istream& is): Settoken()
{
 TRACER("Setsound::Setsound(istream& is) READER");
 read_from(is);
}



Setsound::Setsound(Tokeninfo ti) : Settoken(ti)
{
 TRACER("Setsound::Setsound(Tokeninfo ti) : Token(ti)");

}


Setsound::~Setsound()
{
 TRACER("Setsound::~Setsound()");

}


Token *Setsound::creator(istream& is)
{
 TRACER("Token *Setsound::creator(istream& is)");

 Setsound *newtoken = new Setsound(is);
 if(!newtoken)
 Warning("Memory allocation error during creation of Setsound");

 return newtoken;

}

void Setsound::operator=(const Setsound& d)
{
 TRACER("void Setsound::operator=(const Setsound& d)");
 if(this == &d)
 return;
 Settoken::operator=(d);
 defptr = d.defptr;
}

bool Setsound::operator==(const Token& t) const
{
 TRACER("bool Setsound::operator==(const Token& t) const");
 return Settoken::operator==(t);
}

void Setsound::read_from(istream& is)
{
 string w;

 Settoken::read_from(is); //just read the name

 if(Defsound::table) {
 defptr = Defsound::table->lookup(name);
 if(!defptr) {
 w = "Defsound " & name & " not found in Defsound::table. Using
'default'.";
 Warning(this, w);
 defptr = Defsound::table->lookup("default");
 if(!defptr) {
Warning("Defsound 'default' not defined in Defsound:table.");
 }

 }
 }
 else {
 w = "Defsound::table has not been initialized";
 Warning(this, w);
 }
}

void Setsound::store_on(ostream& os, bool savetype) const
{
 TRACER("void Setsound::store_on(ostream& os, bool savetype) const");

 if(savetype) {
 os << class_desc.name();
 }

 // output a Setsound definition
 Settoken::store_on(os, FALSE);
 os << endl;

}
```

```
Token *Setsound::clone() const
{
 TRACER("Token *Setsound::clone() const");

 return (Token *) new Setsound(*this);
}

void Setsound::display() const
{
 TRACER("void Setsound::display() const");

 if(defptr) {
// the sound system is not fully implemented yet
// soundplay(defptr->soundval);
 }
}


void Setsound::register_token()
{
TRACER("void Setsound::register_token() ");
}

void Setsound::set_name(const string& n)
{
 string w;
 Settoken::set_name(n);

 if(Defsound::table) {
 defptr = Defsound::table->lookup(name);
 if(!defptr) {
 w = "Defsound " & name & " not found in Defsound::table. Using
'default'.";
 Warning(this, w);
 defptr = Defsound::table->lookup("default");
 if(!defptr) {
Warning("Defsound 'default' not defined in Defsound:table.");
 }

 }
 }
 else {
 w = "Defsound::table has not been initialized";
 Warning(this, w);
 }

}
```

APPENDIX C: SAMPLE NPSGDL APPLICATION

This appendix illustrates the use of NPSGDL by showing the source code of a simple GDL application using animation. The following pages contain the program source code and the description files for the GDL objects used by the program. The material and floor object description files are omitted because they are very large.

This application opens a window and displays a simple robot with articulated joints. The robot is built from spheres, cylinders and cones for simplicity. Each of the joints at the shoulders, wrists, hips and knees are controlled by animated variables. The values of the variables is controlled by dials on a button-dial box. This shows some of the interactive capabilities of the GDL system.

The viewing volume for the robot scene is described by GDL camera and viewpoint tokens. The viewpoint uses animated variables to move the viewers location around the robot on a circle. This animated movement is continous and is completely controlled by GDL.

This example is provided to show the ease and power of the GDL system. This program could be extended easily to provide more complex behavior. The source code also serves to illustrate the normal sequence of events required for a typical GDL application.

```
// Sample GDL application robot.C

#include "NPSobject.H" // C++ OFF
#include "gl.h" // SGI gl library
#include "device.h"
#include "Defvariable.H" // To get access to the ruletable
#include "Ruletable.H" // to use the ruletable methods
#include <math.h>

 void initialize();

 // these are global variables updated using the dials.
 // they control the robot joint rotations
 // These variables could be static to a class or something

 float leftshoulder, rightshoulder, leftwrist, rightwrist, lefthip,
righthip,
 leftknee, rightknee;
 // these variables are used to describe the circle
 // the animated viewpoint follows
 float viewangle, viewrad;


 // These are rule functions used by GDL to update Defvariables
 // These are the hooks from GDL to an application to control animation
 // In this case we are returning the current value of interactively
 // modified variables

 float leftshoulderfunc(float f)
{
 return leftshoulder;
}

float rightshoulderfunc(float f)
{
 return rightshoulder;
}

float leftwristfunc(float f)
{
 return leftwrist;
}

float rightwristfunc(float f)
{
 return rightwrist;
}

float lefthipfunc(float f)
{
 return lefthip;
```

```
}

float righthipfunc(float f)
{
 return righthip;
}

float leftkneefunc(float f)
{
 return leftknee;
}

float rightkneefunc(float f)
{
 return rightknee;
}

// these two rules describe the viewfrom circle

float viewxfunc(float f)
{
 return viewrad * fcos(viewangle * .017);
}

float viewzfunc(float f)
{
 return viewrad * fsin(viewangle * .017);
}

main()
{


 short value; /* value returned from the event queue */

 // initialize the ruletable so Defvariables can use rules
 Defvariable::ruletable->add("leftshoulderfunc", leftshoulderfunc);
 Defvariable::ruletable->add("rightshoulderfunc",rightshoulderfunc);
 Defvariable::ruletable->add("leftwristfunc", leftwristfunc);
 Defvariable::ruletable->add("rightwristfunc", rightwristfunc);
 Defvariable::ruletable->add("lefthipfunc", lefthipfunc);
 Defvariable::ruletable->add("righthipfunc", righthipfunc);
 Defvariable::ruletable->add("leftkneefunc", leftkneefunc);
 Defvariable::ruletable->add("rightkneefunc", rightkneefunc);
 Defvariable::ruletable->add("viewxfunc", viewxfunc);
 Defvariable::ruletable->add("viewzfunc", viewzfunc);


 /* initialize the IRIS system open the window etc.*/
 initialize();
```

```
// contains the background color for easy modification
NPSobject backgroundobj("backcolor.gdl");

// the lighting model and light definitions
NPSobject lightobj("light.gdl");

// a huge file containing many Defmaterial tokens
NPSobject materialsobj("allmaterials.gdl");

NPSobject cameraobj("camera.gdl"); // camera and viewpoint definitions

NPSobject floorobj("floor.gdl"); // a checkerboard floor for reference

NPSobject robotobj("robot.gdl"); // the robot description


/* ready the objects for display */

cameraobj.define();

/* define the lights and light model */
lightobj.define();
backgroundobj.define();

/* define the materials */
materialsobj.define();


/* define any lights or materials or other defs
  in the robot
  */
robotobj.define();

/* define any lights or materials or other defs
  in the floor obj
  */
floorobj.define();

while(TRUE)
{

   /* do we have something on the event queue ? process it*/
  if(qtest())
   {

   switch(qread(&value))
   {
      case DIAL0:
         rightknee = value;
         break;
      case DIAL1:
         leftknee = value;
         break;
      case DIAL2:
         righthip = value;
         break;
      case DIAL3:
         lefthip = value;
         break;
      case DIAL4:
         rightwrist = value;
         break;
      case DIAL5:
         leftwrist = value;
         break;
      case DIAL6:
         rightshoulder = value;
         break;
      case DIAL7:
         leftshoulder = value;
         break;
      case REDRAW:
         reshapeviewport();
         break;
      default:
         break;
   } /* end switch on event queue item */
} /* endif qtest() */

   /* draw the background color */
backgroundobj.display();
clear();

   /* turn on Z-buffering */
zbuffer(TRUE);

   /* clear the z-buffer */
zclear();

   /* put up the projection and viewing matrix */
cameraobj.display();

   /* put up the light */
lightobj.display();

   /* draw the floor
   floorobj.display(); */

   /* draw the inputobj */
robotobj.display();

   /* turn z-buffering off */
```

```
 zbuffer(FALSE);

    /* change the buffers ... */
 swapbuffers();

 }
}

void initialize()
{
 /* set up the preferred aspect ratio */
 keepaspect(XMAXSCREEN+1,YMAXSCREEN+1);

 /* open a window for the program */
 winopen("Robot");

 /* make a title */
 wintitle("Robot");

 /* put the IRIS into double buffer mode */
 doublebuffer();

 /* put the iris into rgb mode */
 RGBmode();

 /* configure the IRIS (means use the above command settings) */
 gconfig();

 /* set the depth for z-buffering */
 lsetdepth(0x000000,0x7fffff);

 /* queue the redraw device */
 qdevice(REDRAW);

 // queue the dials and set the value limits
 qdevice(DIAL0);
 qdevice(DIAL1);
 qdevice(DIAL2);
 qdevice(DIAL3);
 qdevice(DIAL4);
 qdevice(DIAL5);
 qdevice(DIAL6);
 qdevice(DIAL7);

 setvaluator(DIAL0,0, 0, 90);
 setvaluator(DIAL1,0, 0, 90);
 setvaluator(DIAL2,0, -90, 0);
 setvaluator(DIAL3,0, -90, 0);
 setvaluator(DIAL4,0, -45, 45);
 setvaluator(DIAL5,0, -45, 45);
 setvaluator(DIAL6,0, -90, 45);
 setvaluator(DIAL7,0, -90, 45);

 /* turn the cursor on */
 curson();

 /* select gouraud shading */
 shademodel(GOURAUD);

 /* turn on the new projection matrix mode */
 mmode(MVIEWING);

}


/************** This is the Robot.gdl description file ************/
/*****************************************************************/
// robot.gdl
/* This file demonstrates the possiblities of the GDL language
   and it's animation support
*/

name robot
// define some variables to be used by animated rotations

defvariable leftshoulderrot
  min -180
  max 60
  init 0
  inc 0.5
  rulename leftshoulderfunc
defend

defvariable rightshoulderrot
  min -180
  max 60
  init 0
  inc 0.5
  rulename rightshoulderfunc
defend

defvariable leftwristrot
  min -55
  max 55
  init 0
  inc 1
  rulename leftwristfunc
defend

defvariable rightwristrot
  min -55
  max 55
  init 0
  inc 1
  rulename rightwristfunc
defend

defvariable lefthiprot
  min -100
  max 35
  init 0
  inc 0.5
  rulename lefthipfunc
defend

defvariable righthiprot
  min -100
  max 35
  init 0
  inc 0.5
  rulename righthipfunc
defend

defvariable leftkneerot
  min 0
  max 90
  init 0
  inc 1
  rulename leftkneefunc
defend

defvariable rightkneerot
  min 0
  max 90
  init 0
  inc 1
  rulename rightkneefunc
defend


defvariable neckrot
  min -90
  max 90
  init 0
  inc 1
  timed yes
defend

// Describe the robot using hierarchical rotations

pushmatrix
// draw the body
setmaterial gold
```

```
defcylinder high
  base 0 9 0
  radius 4
  height 11
  panels 10
endcylinder

// draw the neck and head

setmaterial gray

//  the neck (doesn't rotate)
defcylinder high
  base 0 20 0
  radius 0.5
  height 2
endcylinder

pushmatrix
// the head and nose rotating
  translate 0 24 0
  a_rotate y neckrot

  setmaterial silver
  defsphere high
    center 0 0 0
    radius 3
    panels 100
  endsphere

  pushmatrix
    rotate x 90
    setmaterial pewter
    defcone high
      base 0 0 0
      radius 1
      height 4.5
    endcone
  popmatrix
popmatrix

// draw the left arm
pushmatrix
  setmaterial blueplastic
  translate 5 20 0
  a_rotate x leftshoulderrot
  defcylinder high
    base 0 0 0
    height -5
    radius 1
  endcylinder

  // draw the left hand
  pushmatrix
    setmaterial shinybrass
    translate 0 -5 0
    a_rotate x leftwristrot
    defcone high
      height -4
      radius 1
    endcone
  popmatrix
popmatrix

// draw the right arm
pushmatrix
  setmaterial blueplastic
  translate -5 20 0
  a_rotate x rightshoulderrot
  defcylinder high
    base 0 0 0
    height -5
    radius 1
  endcylinder

  // draw the right hand
  pushmatrix
    setmaterial shinybrass
    translate 0 -5 0
    a_rotate x rightwristrot
    defcone high
      height -4
      radius 1
    endcone
  popmatrix
popmatrix

// draw the left leg
pushmatrix
  setmaterial gold
  translate 2.5 9 0
  a_rotate x lefthiprot
  defcylinder high
    base 0 0 0
    height -5
    radius 1.5
  endcylinder

  // draw the left foot
  pushmatrix
    setmaterial gray
    translate 0 -5 0
    a_rotate x leftkneerot
    defcone high
      height -4
      radius 1.5
    endcone
  popmatrix
popmatrix

// draw the right leg
pushmatrix
  setmaterial gold
  translate -2.5 9 0
  a_rotate x righthiprot
  defcylinder high
    base 0 0 0
    height -5
    radius 1.5
  endcylinder

  // draw the right foot
  pushmatrix
    setmaterial gray
    translate 0 -5 0
    a_rotate x rightkneerot
    defcone high
      height -4
      radius 1.5
    endcone
  popmatrix
popmatrix

popmatrix

/****************** This is the robot camera definition file *********/
/**********************************************************************/
// camera.gdl
/* This GDL files describes a camera and viewpoint used in robot.C
   The viewpoint is animated to follow a circle in the from location
*/

// these variables describe the circle of radius 50
defvariable viewx
 min -50
 max 50
 init 0
 inc 0.3
 wrap no
 rulename viewx
defend
```

```
defvariable viewz
 min -50
 max 50
 init 50
 inc 0.1
 wrap no
 rulename viewz
defend

defpcamera robotcam
nearplane 10
farplane 1024
aspect 1.25
fov 400
defend

defviewpoint robotview
from viewx 17 viewz
to   0 15 0
twist 0
defend

// these tokens are the display portion of this object

// define the viewing volume
loadunit
setpcamera robotcam on
setviewpoint robotview on


/************ This is the robot background color object file *********/
/*********************************************************************/
// backcolor.gdl
// This file defines and uses a color for the robot background
// A GDL file is used to allow easy change of the color

defcolor gray
0.5 0.5 0.5

setcolor gray


/************** This is the lighting description file for the robot ***/
/**********************************************************************/
// light.gdl
/* This GDL file contains the lighting model and light definitions used
   in robot.C
*/

/* define the light for the object
   Lights can have more components than this*/
```

```
deflight whitelight
 ambient 0.0 0.0 0.0
 lcolor  1.0 1.0 1.0
 position 0.0 0.7607 0.7607 0.0
defend

/* define the lighting model */

deflmodel mylightmodel_normal
 ambient 0.2 0.2 0.2
 localviewer no
 attenuation 1.0 0.0 0.0
 twoside no
defend

// This is the displayable portion of the light object

/* turn on the lighting model */
setlmodel mylightmodel_normal on

/* turn on the lights */
setlight whitelight 0 on
```

LIST OF REFERENCES

[Ref. 4] Zyda, Michael J.,Pratt, David R., Monahan, James G. and Wilson, Kalin P., "NPSNET: Constructing a 3D Virtual World", *Proceedings of the 1992 Symposium on 3D Interactive Graphics*, March 1992.

[Ref. 5] Wilson, Kalin P., Zyda Michael J., Pratt, David R., and Monahan, James G. "NPSOFF: An Object Description Language for Supporting Virtual World Construction", in submission.

[Ref. 6] Wisskirchen, Peter ,*Object-Oriented Graphics*, Springer-Verlag, 1990.

[Ref. 7] Wisskirchen, Peter, Obect and Constraint Paradigms for Graphics, "Object-Oriented and Classical Approaches", ACM SIGGRAPH 1991 course 22 notes, 1991.

[Ref. 8] Egbert, Parris K. and Kubitz, William J. "Application Graphics Support through Object-Orientation, November 14, 1991, draft paper.

[Ref. 9] Zelenik, Raobert C., Conner, D. Brookshire, Wloke, Mathias M., Aliaga, Daniel G., Huang, Nathan T., Hubbard, Philip M., Knep, Brian, Kaufman, Henry, Hughes, John F., van Dam, Andries, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques", *ACM Computer Graphics, volume 25 number 4, Siggraph '91 Proceedings*, pp. 105-111, 1991.

[Ref. 10] Struass, Paul S., Carey, Rikk, "An Object Oriented 3D Graphics Toolkit", *ACM Computer Graphics, volume 26 number 2, Siggraph '92 Proceedings*, 1992.

[Ref. 11] Magnenat-Thalman, Nadia, Thalman, Daniel, *Computer Animation, Theory and Practice*, second edition, pp161-162, Springer-Verlag, 1990.

[Ref. 12] SiliconGraphics Computer Systems, Inc., *Graphics Library Reference Manual*, C edition, Iris-4D series, 1990,

[Ref. 13] Gorlen, Keith E., Orlow, Sanford M., Plexico, Perry S., *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons, 1990.

[Ref. 14] *The Texas Instruments C++ Object-Oriented Library Users Manual*, Texas Instruments Inc., 1990.

[Ref. 15] Ellis, Margaret A. and Stroustrup, Bjarne, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Ref. 16] Stroustrup, Bjarne and Lenkov, Dmitry, "Runtime Type Identification for C++, A Proposal for New Features to the Language.", *The C++ Report*, vol. 4 num. 3, pp. 32-42, March-April 1992.

[Ref. 17] Stroustrup, Bjarne, *The C++ Programming Language*, Second Edition, Addison-Wesley, 1991.

[Ref. 18] Shapiro, Jonathan. *A C++ Toolkit*, Prentice Hall, 1991.

[Ref. 19] Coplien, James O., *Advanced C++, Programming Styles and Idioms*, Addison-Wesley, 1992.

[Ref. 20] Silicon Graphics Computer Systems inc., *Graphics Library Programming Tools and Techniques*, pp 2-1 - 2-53, 1991.

[Ref. 21] Silicon Graphics, Inc., *IRIS Universe* issue 11, pp33-34.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Michael J. Zyda Naval Postgraduate School Code CS, Department of Computer Science Monterey, CA 93943-5100	8
4.	Lieutenant Kalin P. Wilson 438 E. 800 S. Salt Lake City, UT 84117	1
5.	David Pratt Naval Postgraduate School Code CS, Department of Computer Science Monterey, CA 93943-5100	1
6.	Lieutenant Comander Don Brutzman Code OR/Br Naval Postgraduate School Monterey, CA 93943-5100	1